DUDLEY KNOX LIBRARY
NAVAL [illegible]UATE SCHOOL
MONTE[illegible]943-5101

Approved for public release; distribution is unlimited.

SOFTWARE REUSE AND THE ARMY PROGRAM DEVELOPMENT PROCESS

by

Donald F. Burns III
Captain, United States Army
B.S., Louisiana Tech University, 1982

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN MANAGEMENT

from the

NAVAL POSTGRADUATE SCHOOL
June 1993

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302 and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1993	Master's Thesis

4. TITLE AND SUBTITLE

SOFTWARE REUSE AND THE ARMY PROGRAM DEVELOPMENT PROCESS

5. FUNDING NUMBERS

6. AUTHOR(S)

Burns, Donald F., III

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release, distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This thesis examines the current Army Program Management software development effort with respect to the application and implementation of software reuse. This study examines current efforts by the Department of Defense and related agencies to implement software reuse into the development and life cycle of both embedded and host application software for automated weapon systems. The DoD software development cycle templates are examined for software reuse applicability, integration, and implementation. Broad overview and analysis of potential, real and perceived reuse implementation inhibitors and barriers is conducted by category (Management, Standards, Library, Legal, and Education), and in conjunction with interviews of critical personnel within the Program Management structure to assess current knowledge and opinion on software reuse. Identified software reuse inhibitors and program personnel concerns are addressed by category, with the intention of finding generalized solutions and application or execution points within the parameters of the software program development structure.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Software Reuse, Reuse Inhibitors, Army Program Manager, Software Metrics	140
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

ABSTRACT

This thesis examines the current Army Program Management software development effort with respect to the application and implementation of software reuse. This study examines current efforts by the Department of Defense and related agencies to implement software reuse into the development and life cycle of both embedded and host application software for automated weapon systems. The DoD software development cycle templates are examined for software reuse applicability, integration, and implementation. Broad overview and analysis of potential, real and perceived reuse implementation inhibitors and barriers is conducted by category (Management, Standards, Library, Legal, and Education), and in conjunction with interviews of critical personnel within the Program Management structure to assess current knowledge and opinion on software reuse. Identified software reuse inhibitors and program personnel concerns are addressed by category, with the intention of finding generalized solutions and application or execution points within the parameters of the software program development structure.

B88-1153
c.1

TABLE OF CONTENTS

MONTEREY CA 93943-5101

I. INTRODUCTION

A. BACKGROUND

The defining difference between weapon systems of today and those of twenty years ago amounts to a single work, software. Computer software, utilized in embedded computer systems, enabled the United States to maintain technological superiority over the numerically superior and once formidable Warsaw Pact and Soviet Armed Forces during the last two decades of the Cold War.

Although this omnipotent force no longer exists, there is still a threat from a multitude of potential enemies. The uncertainty and potential instability in such areas as Southwest Asia and the Commonwealth of Independent States foster a Continuing need for these high-tech systems. Review of the revised DoD doctrine, commensurate with the current down-sizing of the military and future programs, clearly demonstrates that the need for enhanced system performance utilizing highly sophisticated and extremely expensive embedded computer systems will continue to grow. In 1970, the Army inventory contained only three automated weapon systems. By 1980, there were 91. Today, the Army is supporting or developing more than 250 distinct automated weapon systems. [Ref. 1:p. 27] As shown in Figure 1-1, this trend should continue.

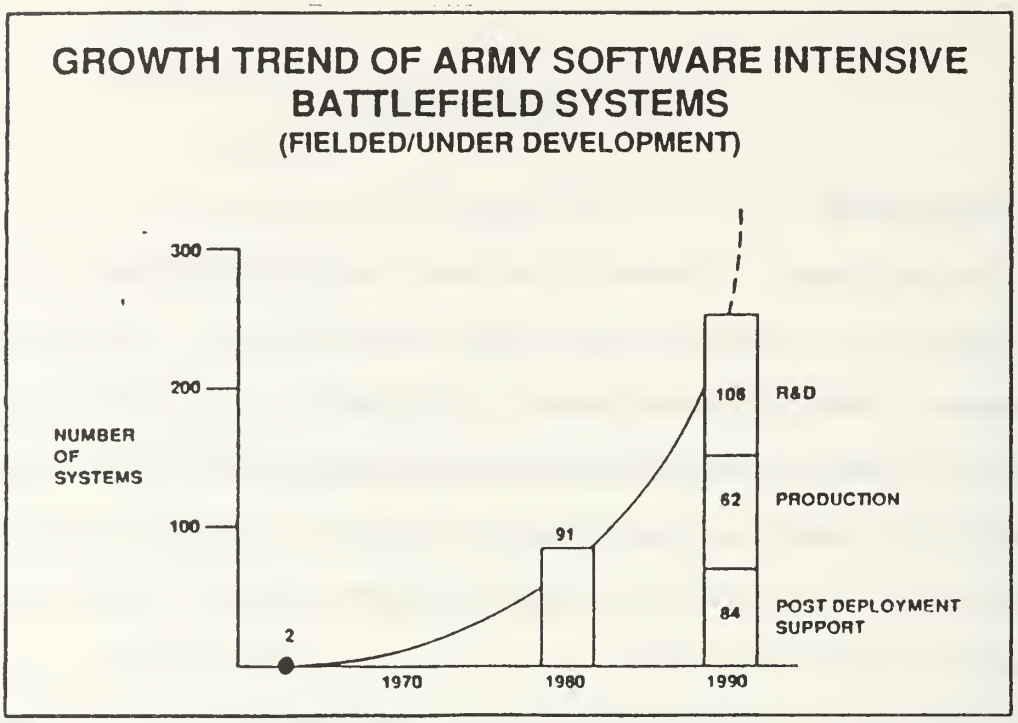


Figure 1-1

Source: [Ref. 1:p. 26]

This technological sophistication is not cheap. In Fiscal Year (FY) 1980, the U.S. Department of Defense (DOD) spent over $3 billion on software. [Ref. 2:p. 19] Most estimates place DOD expenditure for software acquisition during Fiscal Year 1993 in excess of $30 billion. [Ref. 2:p. 19] This represents an approximate growth rate of 12% per year. [Ref. 3:p. 124]

Since 1985 (FY 1986), the budget has contracted to a point where the FY 1993 budget recommended by President Bush provides $63.6 billion for the U.S. Army, a 28% reduction in buying power over FY 90 (as measured in FY 92 constant dollars). [Ref. 4:p. 74] The exponential growth in both the

number of automated battlefield systems and their associated cost has lead to wide discrepancies between the funding required and the funding available for development and fielding of new systems. [Ref. 1:p. 29]

The annual life cycle cost of Army mission-critical embedded software systems is at present in excess of $9 billion, with approximately 30% dedicated to system development and 70% to Post Deployment Software Support (PDSS) (Figure 1-2).

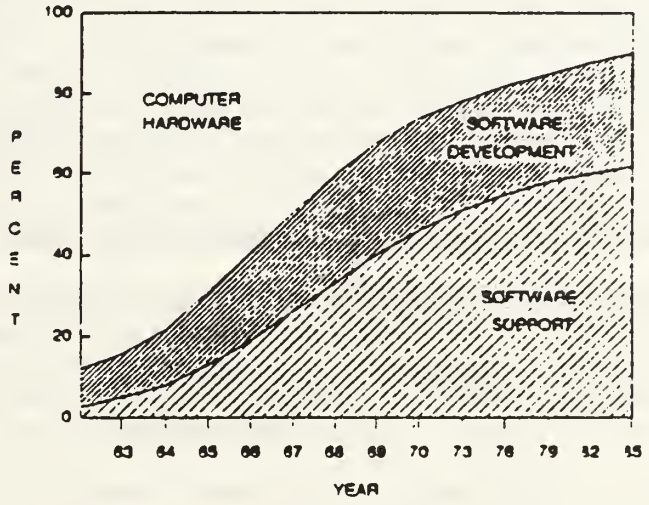


Figure 1-2

Fewer defense dollars, increasing numbers of programs, and rising costs for state-of-the-art weapon systems demand critical analysis of every dollar spent by Army Program Managers to ensure maximum utility and return on investment.

Compounding the financial problem is a decline in software productivity at a time when current software projects are

steadily increasing in size, scope, and complexity. [Ref. 5:p. 3] This decline in productivity is due in part to the aforementioned increase in software system complexity and, in part, to the general decline in the number of available software engineering personnel in the Army, other DoD agencies, and the civilian software development community [Ref. 5:p. 14].

The DoD, in conjunction with a host of corporations involved in developing and utilizing mission-critical program software, has been studying a multitude of proposals to reduce development costs and optimize productivity. These proposals include technical as-well-as non-technical approaches to the problem. Although all areas under scrutiny show potential, the most promising rewards are thought to lie in the area of improved program management and utilization of available resources. In this regard, the application of software-reuse technologies bears great potential value.

B. PURPOSE

The purpose of this thesis is to examine the problems inherent with the application of software-reuse technologies at the DoD Program Office level. The thesis will explore the key aspects of current DoD software-reuse technologies and the principal management inhibitors or barriers to implementation. Additionally, this thesis will query major-systems program managers on their views and observations with respect to software-reuse and its potential or actual effects on their

programs. Finally, this thesis will examine the most promising approaches to resolving the problems associated with implementation of software-reuse within the major-systems Program Offices.

C. RESEARCH QUESTIONS

In accordance with the purpose of this thesis, the primary research question is:

What are the primary problems involved with the proposed software technologies reuse process from the perspective of a Program Office and how might these problems be addressed?

To effectively address this question, the following subsidiary research questions must be answered:

1. What are the key aspects of the DoD software technologies reuse concept?
2. What are the principal management inhibitors or barriers to software-reuse?
3. How do Army major-system program offices view the software-reuse program?
4. What are the most promising approaches to resolving the problems associated with implementation of the software-reuse program?

D. SCOPE

This thesis will limit its scope to the non-technical aspects of software-reuse. Specifically, the potential benefits gained from the implementation of a DoD software-reuse policy governing the use of reuse tools and techniques in the software development portion of major-system program management. It will address only those reuse technologies

currently recognized by DOD and recommended by research groups and advocates for effective, cost-efficient software development. The thesis research will not explore software-reuse beyond the development phase of the program life cycle.

E. METHODOLOGY

The research foundations of this thesis are the documented efforts of the DOD Systems Acquisition and Software-reuse Workshops conducted by the Director of Defense Information of the Office of the Assistant Secretary of Defense, International Electrical and Electronic Engineers (IEEE) Software Seminars and Symposiums, and a host of other Government and civilian conferences and proceedings. Interviews of various personnel within the Army Acquisition Corps Program Office structure provided valuable and otherwise nonobtainable research material. Government reports, publications, minutes and proceedings of current ad hoc software-reuse workshop committee meeting and efforts to develop and implement a reuse process provided additional information.

F. ORGANIZATION

This thesis documents current efforts to implement software-reuse into major-systems program development, and attempts to identify and address potential managerial problems associated with reuse at the Program Office level.

Chapter II reviews the need for and functional nature of software-reuse, some current aspects of DoD software-reuse

programs, and evaluates its current status. Chapter III explains the process used to select program software development methods and the Program Manager's ability to affect software development. The chapter then presents and analyzes inhibitors and barriers to implementing software-reuse. Chapter IV examines the views and observations of Program Office personnel. Chapter V analyzes potential methods for overcoming the problems identified in Chapter III. Finally, Chapter VI answers the research questions, draws general conclusions and provides recommendations for areas of further study.

II. SOFTWARE AND THE DEPARTMENT OF DEFENSE

A. SOFTWARE: DEFINITION AND NEED

The Federal Acquisition Regulation (FAR) defines software as "the set of instructions and data that are executed in a computer. This definition includes only the executable form of the instructions and data." [Ref. 6:pp. 3-5] By definition, software is an intangible, without mass, volume, or other physical properties. It is at best conceptual, lending itself to the nature of an art more than science or engineering. Yet, it is one of the most critical resources of the Department of Defense for the production of today's sophisticated, high-technology weapon systems. The DoD considers weapon systems software to be on the "critical path" of system development. [Ref. 6:p. 1-1]

Software is integrated into virtually every weapon system, either as an integral component, such as an embedded system, or as some sort of training or maintenance complement to the primary system. Today, software has taken the place of many hardware functions in weapon systems. This has become a mater of practical application, allowing system designers much more latitude than previously permitted with strictly hardware or hardwired solutions to advanced problems. Software has allowed weapon systems designers to take advantage of capabilities here-to-fore thought unattainable because of the

physical limitations of man and machine. Software has proven to be inherently more flexible than hardware, and allows both minute performance upgrades and sweeping changes to fundamental weapon systems operations without major computer hardware changes or structural reconfiguration. Such relative ease of change in abilities has proven to be an economic boon to the system upgrade concept. It has proven to be far cheaper and generally quicker to upgrade capability through software changes than system redesign, refit, or rebuild. In addition, it is orders of magnitude cheaper to upgrade than produce new, more advanced weapon systems, as demonstrated by the U.S. Air Force's decision to upgrade existing F16s vice developing a new, multipurpose fighter [Ref. 7:p. A5].

Software upgrading has become increasingly important in this time of similarly rapid evolution of Threat[1] weapon systems' computer technology and associated upgrading. Software has become so prominent in U.S. weapon system design that it has proven to be an influencing factor on overall system design about 50 percent of the time since the early 1970's. [Ref. 6:Ch. 2] However, software as a major component of computerized systems is a relatively new development. It was not until the late 1960's that hardware and software

---

[1]Threat refers to the now defunct Soviet Union, its successor, the Commonwealth of Independent States, various client states of the old Soviet Union, and other potential belligerents.

components of digital systems began to progress along separate paths.

Hardware development has been revolutionary, producing faster, ever more advanced and capable machines. This revolution has been brought about by the marriage of the silicon chip and electrical engineering, and resulted in such things as Very High Speed Integrated Circuits (VHSIC). Today's computers have exponentially increased capability over the earliest practically applied systems. Consequently, this revolutionary process has gone from 16 bit architecture utilizing 60,000 word memories to 32 bit architecture utilizing 5.0+ million word memories. [Ref. 5:p. 2]

Paralleling the hardware revolution has been software evolution. Although a slow process, often being equated to a "black art," software evolution has produced much. [Ref. 8:p. 31] In the roughly forty years since the introduction of the first digital system, software evolution has produced hundreds of languages spanning four levels of complexity[2], multitudes of design and instruction set architectures, and a plethora of development techniques and styles.

Because of the seemingly unlimited ability of computer resources to expand on physical limitations, these resources, especially software, are experiencing voracious demand from

---

[2]Levels of complexity for computer languages are categorized into four groups: machine languages, assembly languages, higher order languages, and application generators. [Ref. 6:pp. 3-10] Each language meets a specific need at a particular level of programming (see Appendix B).

Government and industry. The fact is that tomorrow's weapon systems will be complex technological marvels filled with computer hardware and software.

B. SOFTWARE: PROBLEMS

The sophistication and approach to mission critical computer software design and development provide ample opportunity for problems to arise. These problems take many forms, but generally can be seen to parallel many of the problems that appear in hardware development. Some of the more relevant are as follows:

1. Complexity

As the demand for software grows and the applications expand, the level of complexity necessarily goes up. As program capability expands, especially with regard to real-time systems, the number of source lines of code (SLOC) becomes enormous. Because of the "building-block structure and mind-boggling interrelationships in modern software," such as the number of "do loops" and "go to" instruction sets involved in simple evaluation algorithms, arithmetic software progression can quickly become geometrically complex. [Ref. 8:p. 30] The ambiguous nature and poor decomposition of software problems coupled with the predominant control of projects by "hardware people[3]" expands and aggravates the

---

[3]Hardware as opposed to software people. The term indicates background discipline with respect to the system, such as electrical engineering, aeronautical engineering, etc.

complexity problem. [Ref. 7:p. 4] Technological advances in hardware introduce new requirements into the process, requiring sometimes substantial changes to earlier software packages or modules. Finally, expansion of mission critical computer resources software programs creates a need for substantially larger and more complex associated support systems. [Ref. 6:Ch. 1]

2. Cost

Once, hardware was the "big-ticket" item in terms of weapon system cost. However, over the last two decades, that trend has changed. Today, software accounts for up to 80 percent of the cost of a weapon system (see Figure 2-1). Much of this growth can be attributed to the recognition and inclusion of the costs associated with the entire software system's life cycle. Other sources of cost growth include poorly formulated initial software requirements, upgrading and changing requirements after initiation of the development phase [Ref. 9:p. 50], and otherwise just poor software design methodologies used by some software developers. [Ref. 6:Ch. 1] Add to this the serious decline in real defense spending allocations and appropriations budgeted over the next five years, and the "cost" factor takes on linchpin significance. [Ref. 3:p. 74]

3. Productivity

Nearly every study conducted during the last ten years has given warning to an imminent shortage of software develop-

ment personnel. The current growth rate of the number of software-producing personnel is about 4 percent per year. [Ref. 10:p. 31] Matched against the estimated 12 percent annual growth rate in demand for software and the increasing complexity of today's programs, SLOC productivity is dropping dramatically. As the level of difficulty in "producing lines of integrated, tested, and documented code per month" traverses from "'easy' (precedented) tasks" to complex software tasks, the production rate per programmer drops from about 150 lines per month to 30 lines per month. [Ref. 11]

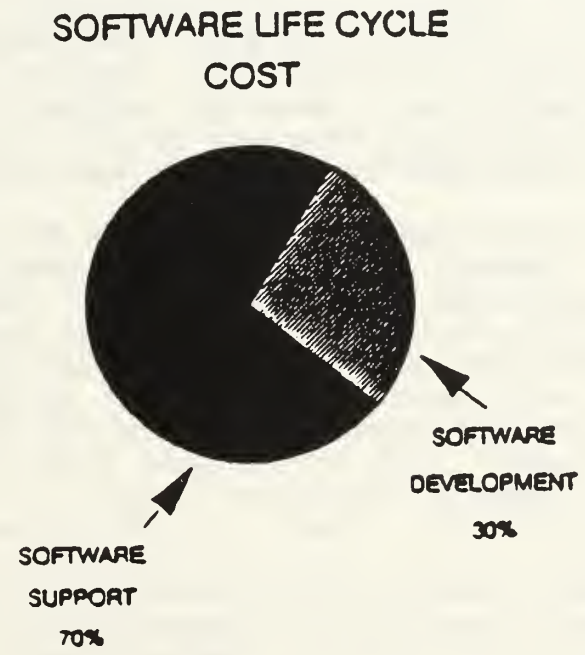


Figure 2-1
Source: [Ref. 2:Ch. 2]

Although the number of programmers is growing, the productivity per programmer is falling. The smaller (relative

to demand) work force and increasing demand will trigger economic responses, driving up the price of labor and the related cost per SLOC. The "bottom line" is simple -- the future portends a dearth of mission critical software.

4. Reliability

Software reliability can be explained simply as the probability that a software component, module, or program will work in a satisfactory manner for a given period of time and under specified conditions [Ref. 12:p. 347]. Reliability is a function of system requirements and program design, and depends on accuracy, consistency, complexity, error tolerance, and modularity. [Ref. 13:p. 229] Predictably, as program size and complexity have gone up, reliability has dropped. Most programs developed to date have not worked as initially designed, and most have never lived up to requirements and design specifications once "repaired." [Ref. 8:pp. 29-30] Software reliability has a direct impact on program cost, often accounting for cost overruns of 50 to 100 percent of total program budget. [Ref. 6:p. 1-1]

5. Quality

Software quality is essentially the absence of spoilage, or that substantial effort (55 percent of the total lifetime cost of the average system) dedicated to diagnosis and removal of faults introduced during the development process. [Ref. 14:pp. 198-200] When eradicating design errors it is necessary to figure both a lost time cost and a

lost productivity cost. The lost time cost consists of the manhour cost of repairing the unfruitful labors of previously expended and expensed manhours and the utility cost of not having a fielded system. The lost productivity cost, while only an estimation, must be extremely high given that efforts to repair one piece of software must necessarily pull resources away from the production of new software, aiding in the overall decline in software productivity.

C. SOFTWARE: SOLUTIONS

Recognizing that no single program, project, or group can "fix" all of the problems listed in Section B, the Army, in conjunction with the DoD, has approached these problems with various solutions. Some of the more important or influential are as follows:

1. Studies, councils, and working groups

The DoD has engaged in and contracted for a number of studies and working groups to address the problems and concerns involved with mission critical computer resources development and acquisition. [Ref. 15:p. 1] Efforts have been directed at almost every aspect of these problems. They have involved studies of weapon systems' software management, cost-schedule strategies for software intensive project development, the software development environment, and technology transfer. [Ref. 16:pp. 4-5] Additional areas of focus have been on issues involving software reuse technologies, including the Software Technology for Adaptable,

Reliable Systems (STARS) Program. [Ref. 17:p. 1] There have also been policy and standardization attempts made to address problems within the segment of the Defense Industrial Base responsible for most DoD software production. [Ref. 18] And finally, a great variety of joint Service attempts have been made to apply standardization to shared common concerns such as avionics hardware and software and command and control communications systems. [Ref. 6:Chs. 3-4] [Ref. 19:p. A-22]

Most notable of these efforts has been the great proliferation of permanent panels, working groups and software study organizations. Also of note has been the number of non-DoD organizations and civilian companies which have also formed to advance computer technology, regardless of the apparent beneficiary of their findings and proposals. Some of the more prominent and important of these organizations and companies are:

(a) Defense Advanced Research Projects Agency (DARPA)

(b) Software Engineering Institute (SEI)

(c) Defense Information Systems Agency (DISA)

(d) Joint Logistics Commanders (JLC) Joint Policy Coordinating Group on Computer Resource Management

(e) Institute for Defense Analyses (IDA)

(f) National Security Industrial Association (NISA)

(g) Joint Integrated Avionics Working Group (JIAWG)

(h) U.S. Army Communications-Electronics Command (CECOM) Center for Software Engineering

(i) NASA Langley Research Center

2. Standardization

Most of the problems listed in Section 2 are the result of the uncontrolled proliferation of the number of weapon systems utilizing computer applications that occurred from the late 1960's through the 1970's and into the early 1980's. [Ref. 6:Ch. 4] This multiplicity of problems posed by the non-standardization of supposedly interoperable systems reached a peak in the late 1970's and prompted the DoD to focus on standardization, particularly with regard to embedded systems. Through the efforts of most of the groups listed above, and especially the agenda carried by the Joint Logistics Commanders and DARPA, significant advancements have been made in both hardware and software standardization.

To this end, the DoD has standardized nearly every aspect of software and hardware development, production, and procurement through issuance of specific policy and guidance. The primary governing directives is DoD Directive 7920.1, Life Cycle Management of Automated Information Systems [Ref. 21]. Three areas have received the primary focus:

(a) Higher Order Languages (HOLs). As mentioned earlier, there were hundreds of languages being used within the DoD by the late 1970's. In order to inject some much needed interoperability into embedded systems and limit geometrically expanding software support costs, the DoD severely limited the number of acceptable HOLs and designated Ada as the preferred language. [Ref. 6:Ch. 4]

While DoD Directive 3405.1, Computer Programming Language Policy, lists and provides guidance for selection of the approved DoD programming languages [Ref. 22], DoD Directive 5000.2, provides explicitly

for the use of Ada as the single, common, HOL in new computer embedded weapon systems [Ref. 20:p. 6-D-3].

(b) Software Development. Technological advances and language standardization aided software development immensely. The insertion of Very High Speed Integrated Circuits (VHSIC) has provided much greater potential application of the acceptable software languages. to exploit this potential while maintaining control of resources, the DoD has implemented DoD-STD-2167A, Defense System Software Development, and DoD-STD-2168, Defense System Software Quality Program, to apply standardization and a system engineering approach to software development. [Ref. 23] Additional efforts, such as the STARS program and liaisons between technical and academic institutions have aimed at fostering technology application and transfer [Ref. 6:Chs. 4 - 5].

(c) Computer Hardware. Because the revolutionary developments in mission critical computing hardware took as many directions as there were software languages, the DoD expended considerable effort standardizing the Instruction Set Architecture (ISA). This standardized the "Internal and fixed repertoire of Instructions" that compose the required roadmap describing the hardware/software interface. [Ref. 5:Ch. 3] Although an ISA is currently established as MIL-STD-1750A[4], the tremendous pace at which computer technology is moving forward has rendered this architecture obsolete. [Ref. 26:pp. 36-37]

DoD attempts at regulating the computer development environment through directives, regulations, and standardization have successfully surmounted, or at least curtailed, many of the problems described in Section 2. Often the proposed solutions for a particular problem produce "bleed-over" into other problem areas, directly or indirectly complimenting

---

[4]The currently established MIL-STD-1705A, 16-bit architecture, is generally utilized in airborne applications and is probably the most widely used system. [Ref. 6:Ch. 3] However, other standard-ized 8-bit and 16-bit architectures do exist within certain communications applications. [Ref. 24] [Ref. 25]

other efforts to address problems in any given area. For example, the standardization and implementation of MIL-STD-1705A architecture fostered the development of the Ada HOL, which in turn has enabled the DoD to more effectively structure the software development environment. Obviously, as each problem exerts some influence on the other problems individually and collectively, so too do the proposed and potential solutions.

This iterative process of problem evaluation, solution application, problem evaluation, is slow, ponderously expensive, and yields only marginal results. Recognizing the need to optimize efforts to solve mission critical computer resources problems, the DoD and its many supporting and allied agencies have attempted to utilize all the tools available to maximize computer resource potential. One of the most viable solutions, widely ranging in terms of impact and potentially lucrative in terms of results, is application of software reuse technologies and methodologies. The potential feasibility of this has been borne out by studies showing as much as 40 to 60 percent of the software written for one system is virtually identical to previously written code for a similar system [Ref. 2:p. 20].

D. SOFTWARE REUSE: TECHNOLOGIES AND METHODS

DoD has settled on the definition of software reuse as being any new application of an existing component to include requirements, designs and specifications, and final source

code, as well as corresponding test plans, procedures, results, and supporting documentation, generated during any stage of a system's development. [Ref. 27] This definition opens the scope of software reuse to much wider application than the DoD definition of software (Section A) would suggest. This definition facilitates implementation of reuse not only from the technical aspect, but the all too long ignored non-technical aspect.

While the concept of software reuse has been around since the very first digital computers, reusability did not become a major topic within the computer industry until 1983 [Ref. 28:p. 372], even though the DoD had initiated efforts through the JLC to implement reusability as early as 1981 [Ref. 15:Tab E]. Although 1983 marked the beginning of real efforts, from a technological perspective, to develop the mechanics of software reuse, only in the last three to four years has real interest been shown in examining and developing the non-technical side of software reuse.

To understand the non-technical aspects of software, it is first necessary to have at least a rudimentary knowledge of the technical side of software reuse. The first step in development of new software from existing software is domain analysis. This is a process wherein preliminary requirements for software parts are identified to fill common needs within the specified domain. The process develops a preliminary domain model and a classification scheme. Then, through the

collection, organization, and analysis of the data, the model is refined to identify common objects, structures, and functions as candidates for reusable parts. [Ref. 29:p. 1] The domain analysis is independent of the type or application of reuse. [Ref. 16:p. 5]

The second step requires a thorough cost analysis to ensure there is a benefit to be gained through reuse. A relatively simple formula has been developed by the JIAWG based on the "first-cut" domain analysis and estimated potential for reuse across the system domain. The formula and process [Ref. 29:pp. 5-11] are as follows:

(1) Determine overall estimation of potential software parts reusability. Base the estimation on high, medium, and low ratings of software parts and areas, with high being greater than 50 percent and low being less than 30 percent reuse of products.

(2) Assumptions:

(a) Software parts (e.g., all documentation, design representations) are reused.

(b) There will be some cost associated with developing reusable software to current project standards.

(c) The more times a software part is reused, the lower the cost of that part.

(3) Let:

B = relative cost of locating and integrating reusable software parts ($0 < B < 1$).

R = proportion of reused software parts (proportion of new software parts = $1 - R$).

E = cost to develop reusable software parts relative to non-reusable parts [includes development of library and standards ($E > 1$).

C = relative cost of developing the total project software parts (break even point: C = 1).

N = estimated number of times code must be reused to break even [N = E / (1 - B)].

(4) Then:

C = (1 - R) * 1 + R * (B + E/N) or

C = (B + E/N-1) * R + 1

As stated earlier, this is a rather simplistic approach. However, the point is made graphically (based on a "best case estimate" in which "software parts up to and including code are reused" [Ref. 29:p. 11]) in Figures 2-2 and 2-3. Figure 2-2 demonstrates the percentage of cost savings for different values of B. The lower the value of B, the more that is saved. Figure 2-3 demonstrates potential cost savings based on various values of reusable software parts. In this graph, the relative cost of developing reusable software parts is 1.25 and the relative cost to reuse it is 1.0. [Ref. 29:pp. 9-11] These graphs demonstrate the potential for reuse. However, while on the surface this would appear to be a simple matter, the potential technical difficulties of utilizing still immature techniques for reuse can quickly overwhelm any cost advantages. Obviously, thorough domain analysis becomes even more crucial.

Should both steps prove advantageous, however, there are two different approaches to successful implementation of software reuse. The two approaches are based on either "the

COST IMPACT OF REUSING SOFTWARE

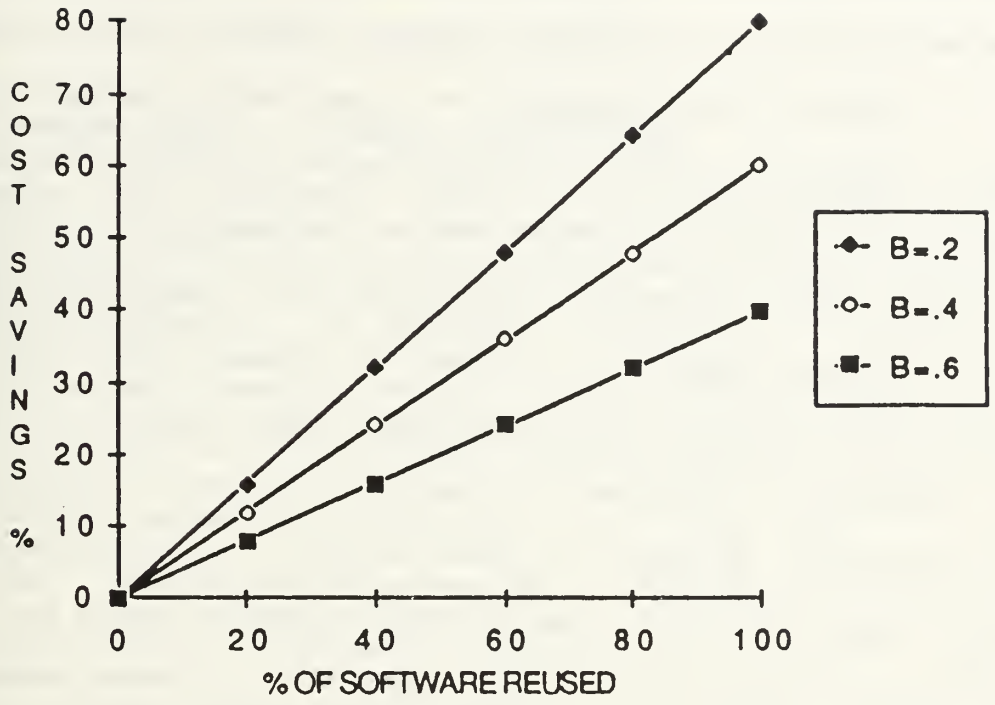


Figure 2-2

COST IMPACT VS. USE
E=1.25, B=.1

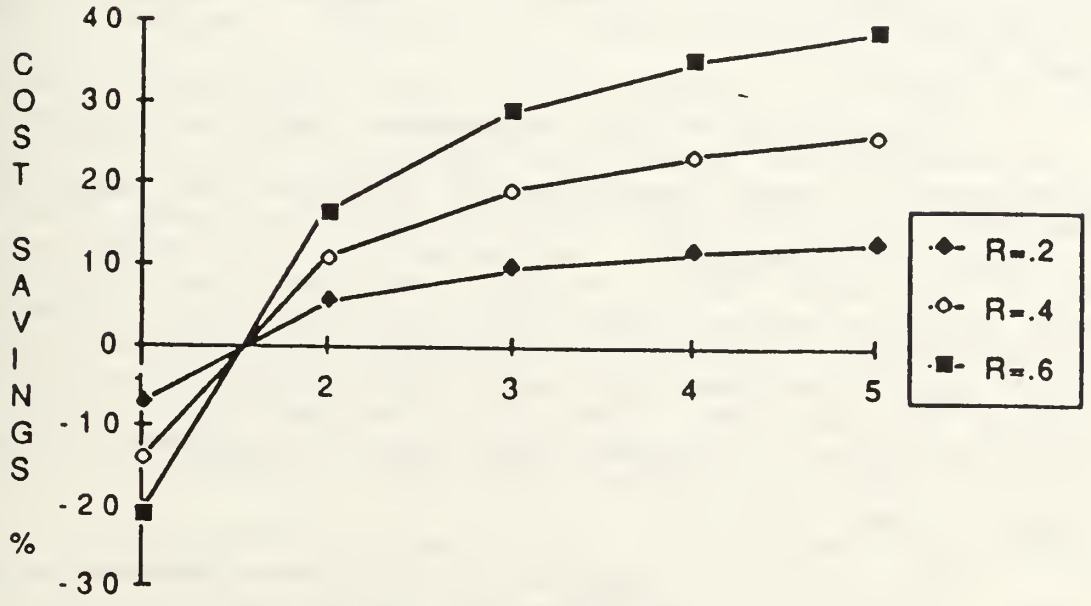


Figure 2-3

origin and packaging of the component, or the granularity of components that are reused." [Ref. 16:p. 5].

1. Origin and packaging of components refers to the system commonly detailed as the software library or repository system. The library system can be founded on a variety of precepts, from libraries containing only components within certain domains (generally indicating intense domain analysis) to libraries stocked with individual coded lines of software to libraries consisting of combinations of reusable data, architectures, designs, software modules or entire programs. [Ref. 28:pp. 372-376] The retrieval system works much like any other resource library, with indices and cross references based on such things as keywords, domain types, architectures, etc. The repositories, on the other hand, do not generally classify and catalog software parts. They generally "have no order or commonality and usually no controls are exercised." [Ref. 29:p. 6]

2. Granularity of the components, such as code reuse, specification reuse, generation of software, and reuse based on generic architectures, is the other way to define reuse. This involves taking actual code, specifications, architecture, etc., from one application and using it "as is" or modified for use in another application, regardless of system design. [Ref. 16:p. 6]

The choice of approach to reuse then depends on such variables as the domain, its boundaries and level of

technological sophistication, expertise in the field and mission of the system. [Ref. 16:p. 6]

E. SOFTWARE REUSE: CURRENT APPLICATIONS

There are currently several on-going software reuse efforts, both within the Federal Government and by industrial programs supporting the Federal Government. Because of software reuse's need to access preexisting software parts, efforts limited to a particular military Service, university or industry software engineering community tend to limit potential. These current programs encompass the efforts of the DoD, National Aeronautic Space Administration (NASA), several universities, and a number of software engineering contractors. Some of the more significant software programs involving reuse follow:

1. Software Technology for Adaptable, Reliable Systems (STARS) Program. STARS is a major software enhancement program directed by DARPA. The program, initiated in the early 1980's, with the mandate to build on the achievements of the Ada program, is aimed at improving the software development and support environment. [Ref. 30:p. 10] It was designed to cover both technical and management aspects during all phases of the software life cycle, and is part of a joint effort with the very high speed integrated circuit (VHSIC) program to improve management practices, software acquisition strategies, technologies, and personnel skill levels. [Ref. 31] The STARS program is "trying to increase software

productivity, reliability, and quality by integrating support for software processes and reuse concepts." [Ref. 16:p. 5] Currently, STARS is sponsoring two programs focused specifically on reuse:

(a) **ASSET.** Asset Source for Software Engineering Techniques. This program is focused on developing and exploiting technologies to permit effective interoperability between reuse libraries. [Ref. 16:p. 5]

(b) **CARDS.** Central Archive for Reusable Defense Software. The CARDS program is tasked with "creating a knowledge blueprint" specifying "how to build domain-specific reuse libraries." [Ref. 16:p. 5]

2. There are several software library programs which are classified as "origin of component: programs:

(a) **RAPID.** Reusable Ada Products for Information System Development. This is an Army program with objectives to promote "reuse" of Ada software components and reduce systems development and maintenance costs. It utilizes an automated system for identification, analysis, storage, and retrieval of Ada reusable software components, including source code, requirements, and design criteria. [Ref. 32:pp. 31-32]

(b) **CAMP.** Common Ada Missile Packages. The CAMP program is sponsored by the U.S. Air Force Armament Laboratory and is operated by McDonnell Douglas Missile Systems Company. It serves the military, NASA, and civilian contractors developing missile systems software. This reuse program consists of taxonomy classified software packages aimed at applications in a real-time domain. [Ref. 33]

(c) **AFATDS.** Advanced Field Artillery Tactical Data System. The AFATDS program provides a library system within the Army Tactical Command and Control System (ATCCS). This library is designed to provide reusable Ada software components for fire support applications and to have interface commonality with the other battlefield functional areas (BFAs)[5] within ATCCS.

---

[5]The five Battlefield Functional Areas are Maneuver Control, Fire Support, Air Defense, Intelligence and Electronic Warfare, and Combat Service Support.

(d) **Eli.** Eli is a NASA sponsored library facility. It is a "knowledge-based reusable software synthesis system" designed to classify, store, and retrieve software as well as create an environment that "emphasizes, encourages, and supports reuse." [Ref. 34:p. 17] It is intended to be part of a system incorporating the software development tool CASE (Computer-Aided Software Engineering system) and the Architecture Design and Assessment System (ADAS) with the goal of automated system development. [Ref. 35:p. 65]

(e) **AdaNet.** This is a project under the direction of the NASA Johnson Space Center. The AdaNet objective is to develop an electronic distribution network for software engineering information, parts, and code. The program serves the U.S. Government and private industry working on software development in manufacturing and administrative areas. [Ref. 36]

3. There are far fewer programs which utilize the "granularity of components" approach to software reuse. However, one of the largest and most complex projects is the Army's ATCCS project (see D.2(c) above). The Army Tactical Command and Control System seeks to tie together command and control systems being developed by the five BFAs. Being developed from commercial non-developmental (NDI) computer systems and commercial and Governmental off-the-shelf software (COTS and GOTS), ATCCS employs specification reuse at the "A", "B", and "C-5" specification level. Additionally, the program utilizes a generic architecture to provide high-level design for related applications within the five BFAs, and provides for planned and potential future technology insertion. [Ref. 37]

F. SUMMARY

Obviously, a great deal of time and effort are being spent on reuse application. The process is extremely complex, blurring the boundaries between the technical and managerial aspects of program development. It requires the software developer to consider the managerial aspects of cost and development time while the program manager must consider reuse methodologies and available resource pools. It requires long term investment and provides dubious returns, especially in the short run. However, it appears to be one of the best answers to the software program issues of increasing demand and productivity deficit, short of halting technological advance. This chapter addressed software definition, problems, solutions and reuse technologies. The next chapter will focus on program management software development methods and the barriers to implementing a software reuse program.

III. SOFTWARE REUSE

A. THE PM AND SOFTWARE DEVELOPMENT

Regardless of the type of software needed for a system, the eventual product can be arrived at only after the development process has run its course. The program manager will determine the direction and ultimate end of that course. It is within the venue of the program manager to influence nearly every aspect of software development.

The PM's level of technological sophistication, comprehension of software, software architecture, and software engineering concepts, and overall familiarity with the software development process, will in large part, determine the shape of the final product. The complexity and sophistication of the final software product, whether embedded or not, will be determined by the degree and depth of involvement displayed by the PM.

This concept is very simple. The greater the PM's knowledge of cost estimation, system requirements, software architecture, design, engineering, and test and evaluation, the more he is able to influence the software development process. The less the PM knows, the more likely he will rely on subordinates or outside contractors to make critical decisions and influence development. Therefore, the final product will be the result of personal bias and preferences of

either the PM or knowledgeable subordinates or contractors tasked with software development or management. This can be reflected in something as simple as the choice of an off-the-shelf product such as SCO UNIX[6] to be used as an operating system with the Army's Tactical Command and Control System or as complex as the embedded systems found in cruise missiles utilizing satellite global positioning system navigation programming. It is easy to understand that a developmental program will be more susceptible to, and indeed be more reflective of, PM influence than a COTS program.

B. THE SOFTWARE DEVELOPMENT PROCESS

To understand the problems facing the Program Manager in developing software, let alone attempting to incorporate software reuse into the acquisition process, it is critical to understand the product development cycled or process. Only when this process is thoroughly understood can attempts at schedule reduction and cost saving through reuse be attempted.

The development cycle or process for software is similar to that used for hardware with only a few exceptions. One difference is the idea that software development is only the first part of the software life cycle, whereas the hardware life cycle is generally acknowledged to begin after the

---

[6]SCO UNIX is typical of a commercial off-the-shelf host operating system for standard applications programs. SCO is the brand name and stands for Santa Cruz Operations while UNIX is the type of system, much the same as MS (Microsoft Systems) DOS.

development process is completed. The most obvious difference however, is the end product. Hardware development results in the production or completion of some physical product which performs a visible or measurable function. There is a definitive end to the process, a tangible finality to the development effort. For hardware, the end of the development process marks the beginning of the production process. Software on the other hand is more abstract in its end result. There is no physical product, and performance can be judged only after extensive testing. Production is merely a matter of copying the final product to other disks for use in other machines, or embedding the software onto silicon chips integrated into the system. And because there is no physical product, it can be difficult to determine a definitive completion point. This often leads to distortion of the programming as the project attempts to reach completion.[7] The urge to "add"capabilities and enhancements after the development phase has been initiated, combined with the complexity of today's programs and difficulty in determining the end point

---

[7]Developing computer software is much like taking a trip from point A to point B by following a road map. There are several routes that can be taken -- the direct route (straight line) and any number of more circuitous routes. Even after the journey starts, it is possible to veer off the most direct route (for any number of reasons) and add miles to the trip. The same concept applies to software. Although there is not much in the way of a road map to follow, there are infinite detours and "scenic routes" in the programming which provide no added value to the product and may even complicate or delay program completion.

requires the software developer and program manager to exercise close-hold management over the process.

Whether the final product is embedded application software (generally the ultimate goal of the software acquisition activity), commercial off-the-shelf software bindings, or development support, maintenance, and diagnostic software, the program manager follows one of three development process models. The conventional or "waterfall" software development model is shown in Figure 3-1, and evolutionary offshoot is presented in Figure 3-2, and finally the prototyping approach is depicted in Figure 3-3.

The conventional software development model, commonly referred to as the "waterfall" software development process because of the way it is graphically presented, is the baseline model. It is the system most often used to manage DoD software development. To understand the evolutionary and prototype development processes it is critical to understand the waterfall method.

The steps in this standard software development process are as follows: System Definition, Software Requirements Definition, Preliminary Design, Detailed Design, Code and Unit Testing, Integration Test, System Test, and Maintenance. [Ref. 38:pp. 19-20]

Figure 3-1. Conventional or "Waterfall" Software Development Process

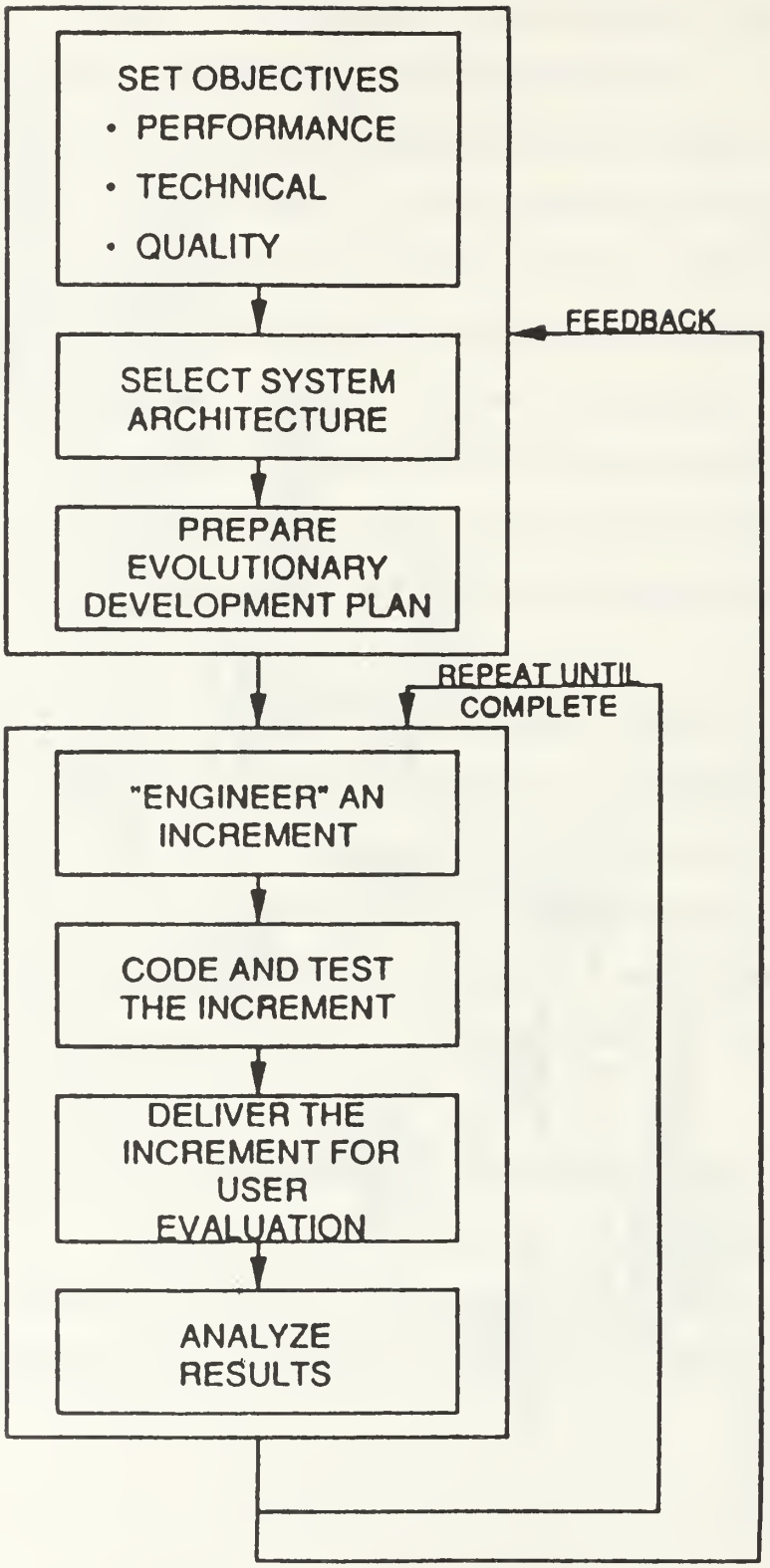


Figure 3-2. Evolutionary Software Development Process

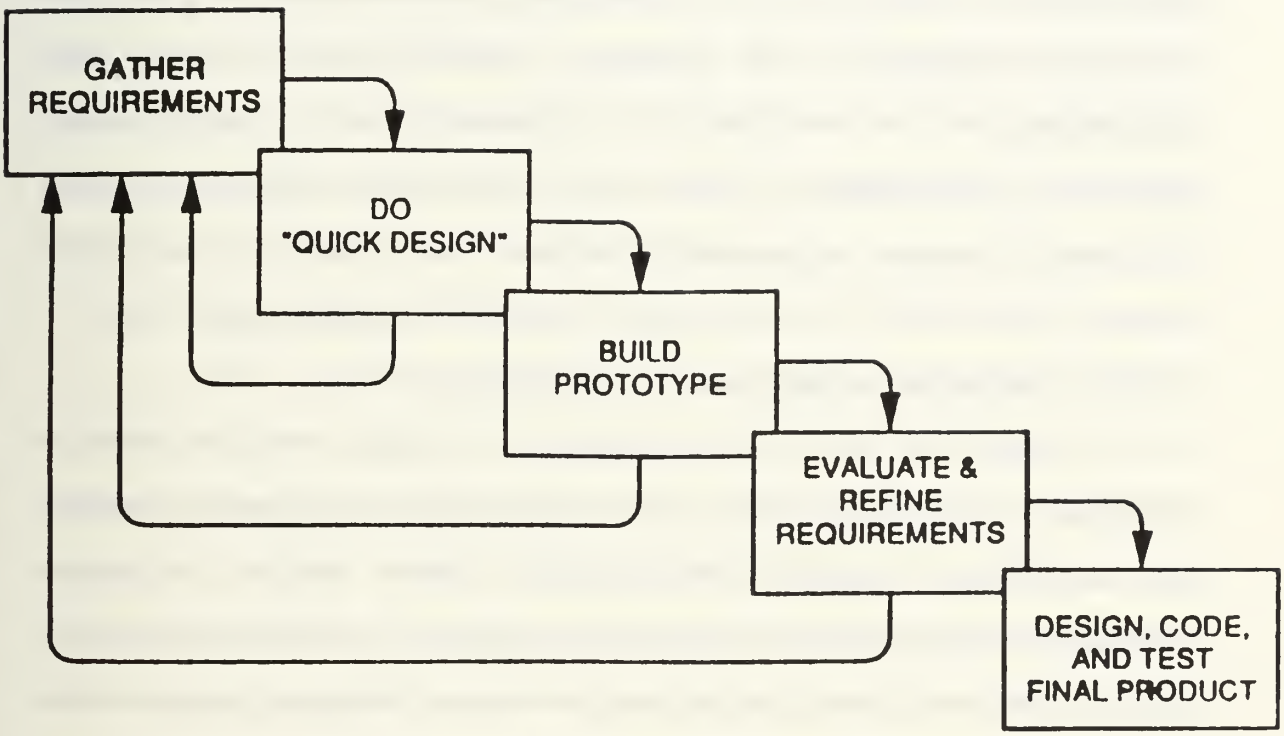


Figure 3-3. Prototyping Software Development Process

1. System Definition

The first step actually happens prior to the start of the development effort. It starts with the definition of the system architecture. This is the allocation of system requirements to either hardware or software components. Once the decision is made as to which functions will be executed by hardware and which will be executed by software, a parallel effort of hardware and software definition, design, and implementation is initiated. Although there are now two separate development paths, the two must remain linked together through cooperative effort to ensure successful system integration is the final product. [Ref. 38:p. 20]

2. Software Requirements Definition

This is the most important step in the entire development process. Insufficient requirements identification and definition create development errors which propagate through the development process, to end in erroneous hardware designs and software products which can be extremely expensive and time consuming to correct (if they can be corrected at all). Therefore, this step which uses the System Specification as a guide to establish the requirements for each computer software configuration item, including software inputs and outputs, interfaces, and controls, is crucial in the development process.

As mentioned earlier, testing, based on the requirements, is the method used to determine if the product has

reached completion and if it satisfies the system needs. The testing criteria are based on the requirements definitions. Consequently, it is possible to poorly define, incorrectly develop, successfully test, and ultimately produce a product that does not meet system specification.

It is often necessary to develop testing tools, modules, and interfaces into individual modules concurrently with the program software in order to be able to test the final product. These testing tools add even greater potential for error and unsuitable end product. Again, the importance of correctly defining the requirements comes into focus.

3. Preliminary Design

This activity determines the overall software structure. The software is partitioned into modules based on the requirements identified in the software requirements definition phase. The function (or functions) of each module is then defined, as-well-as the relationships between modules. The software executive functions which contribute the timing and priority rules for the software structure are also defined at this stage. To ensure the software requirements meet hardware constraints, the timing and memory budget for the software tasks and modules are also established during this phase.

Currently, there are two types of software design methods -- functional development and design-oriented development. Both engage in the decomposition of the system.

One is based on the concept that each module represents a major step in the overall process. The other is based upon the concept that the behavior of an object, such as a disk driver or video display, is characterized by the operations involving or performed by or on that object, and the functions it requires of other objects (such as the demands made of the SCSI driver by an external peripheral). [Ref. 38:p. 22]

Again, as in many software engineering operations and procedures, both these methods are hierarchical. They exist as building blocks, with the smaller blocks of the lowest level forming the larger blocks of the next higher level and so on. Ultimately, the top-level modules can accomplish their application or execution tasks with the culmination of all the subtasks.

In its most effective form, the average software module contains about 100 lines of source code and seldom exceeds 200 lines. Each of the blocks or modules in the hierarchical structure should generally contain more than two but less than seven smaller blocks or submodules to maintain an effective span-of-control over the function.

4. Detailed Design

Having broken down the overall program into manageable modules in the previous phase, this step proceeds to detail the design of each software unit module. The detail is expressed to such a point that the coding sequence can be done by someone other than the software designer. This requires

that the detailing include the unit's function, inputs, outputs, and memory and timing constraints, as-well-as the logical, static, and dynamic relationships between modules. It is critical, therefore, that the completed detailed design include descriptions of all data to be processed and all processes to be executed. Finally, this step generates the module and system integration test specifications and procedures.

5. Code and Unit Testing

In this step, the detailed software design is translated into the machine code of the target computer and, when complete, tested to eliminate errors.

The coding process involves writing source code for each software component or module in a Government approved higher-order languages such as Ada. A compiler is then used to translate the source code into object code.

As with the previous steps of the software development model, the relative ease of completion and degree of success of the coding phase depends directly on the software requirements and documentation produced in the earlier steps. However, an even more crucial factor can affect the outcome of the coding sequence at this point -- the human factor. Because each unit or module of higher-order code is written by a human programmer, the greatest potential for errors in the coding sequence comes from that programmer.

The unit testing of the software generally detects most of these coding errors, which are then corrected by the programmer. Errors that go undetected during this phase will usually turn up during the next step, software module integration testing.

6. Integration Test

This step entails combining the small, manageable, encoded modules into larger modules and testing the integration to ensure compliance with system requirements. Obviously the module integration must usually be based on the functional capabilities demonstrated by groups of specific units. The integration is done incrementally, with coded units being added only after each assembled sub-module is successfully tested.

Because the integration testing involves several coded units, it can be extremely difficult to locate and isolate faults or errors. Fault isolation and correction can be a tedious, expensive, and time consuming business. The importance of a well designed software test plan and test description is clearly evident.

7. System Test

This is the final step in the actual development process. This step demonstrates that the hardware works in conjunction with the software to accomplish all system functions. Successful completion of this step will result in a finalized product ready for production.

8. Maintenance

The Maintenance portion of the developmental process carries the software through the rest of its life cycle. If properly executed, the previous steps result in a product that is readily maintainable and thoroughly documented. It remains only for the maintenance process to correct any residual problems in the software, adapt the software to hardware or operational environment changes, and enhance the software to enhance performance or incorporate new functions. [Ref. 38:p. 25]

Although, this sounds relatively simple, the maintenance phase of the life cycle is actually a process in and of itself. Software maintenance is the most costly step in the overall process, as the life of the product can extend for years.

It is possible for errors in the software to present themselves long after the software has been released for use. These problems can result form subtle changes or differences in the host hardware, expansion of the intended application environment, or interaction with other software. However, a large number of these errors can be attributed to poorly designed tests developed in parallel with the software. Poorly designed and developed unit or module tests as-well-as integration tests will plague software throughout its usable life. The error correction process, as described previously,

is a fairly straight forward (albeit difficult) matter of fault isolation and program correction.

The process of inserting changes in the programming to accommodate a changing operational environment or minor hardware changes, however, is a small scale repetition of the software development process. The new requirements are identified and documented. The change(s) to meet the new requirements must be designed within the parameters of the existing software design. The design is then coded, debugged, and tested as in previous steps. It is critical, given the influences of changing requirements and the nature of inserting new code into the existing program, that exacting and deliberative testing be conducted to ensure the new code does not adversely affect or impact the already existing code. The only deviation from the original development model is the addition of an end review at the completion of the change process to ensure the new requirements are met and the unchanged functions still operate correctly.

The other two methods mentioned previously are used in cases where the waterfall method may not apply because of difficulty in defining the software requirements fully at the beginning of the process; significant risk created by the design approach to satisfy requirements; or the user requirement for early initial capability (utilization of software). [Ref. 38:p. 15]

Although the evolutionary development process can be applied to any development effort, it is best applied to very large software development projects or when intermediate software products are necessary or required. The evolutionary model reduces large programming efforts into manageable size, then breaks the system requirements into phases and applies the waterfall life cycle to each phase. This method allows the positive demonstration of evidence that the final product will work effectively.

The prototyping approach is designed to bypass the normal software documentation in favor of speed of development. Instead, a prototype program is developed to demonstrate proof of concept. Once the concept is demonstrated, the prototype program is discarded and the standard waterfall development process, including the production of documentation, is followed to produce the final software product. Because the prototype software program has no documentation, it is virtually unmaintainable, and therefore of little use except to prove the required concept.

The development process, as detailed above, is ideally suited to implement software reuse at the program or project level. The actual reuse implementation process is much like the software life cycle maintenance process. However, this is not to imply that the reuse process can be integrated into the process just anywhere.

Recalling that the definition of reuse includes any new application of existing components (i.e., requirements, designs, specifications, source code, and test plans, procedures, results and documentation) the opportunity for utilization software reuse methods is quite obvious.

C. OPPORTUNITIES FOR REUSE

The first step in applying reuse to any new system is to examine the operational domain of the system under scrutiny. In essence, if the system to be upgraded or developed is a missile, then other missile systems should be considered as candidates for potential reuse contributions. Other areas which should be targeted for review for reuse application should be systems which incorporate a particular characteristic or functional capability identified as being conceptually necessary or required in the concept exploration phase of program development.

During the System Definition phase, a close examination should be made of existing architectures in similar systems. Reuse requires that these architectures be examined for advantages and opportunities to incorporate both hardware and software technology into the new system. Although fielded systems generally contain less than state-of-the-art or leading-edge technologies, reuse of existing architectural concepts preclude large expenditures on concept exploration -- essentially eliminating the re-invention of the wheel. Even something as seemingly minor as the approach used to develop

the architecture in an existing system can be utilized to save time and money in system development. Any reuse of architecture will necessitate further investigation of the existing system for reuse of other components.

Reuse of existing systems specifications and associated documentation can provide baseline requirements for new systems as-well-as systems undergoing upgrade. If nothing else, a comprehensive review of requirements for currently fielded similar systems should highlight shortcomings in requirements identification and definition. Test procedures, standards, and results of similar systems should be examined, again with the idea of baselining. As mentioned earlier, testing is critical and can lead to devastating results in the end product when incremental testing shows the system development to be on target, but the final product falls short of overall performance specifications. Only after a project is complete and undergoing operational testing does it become apparent that the system is inadequate, usually requiring more time and money to fix the problems, if they can be fixed at all. Reuse can potentially save considerable time and money if applied at this stage of the software development process. Even if actual requirements and specifications from other systems are not reused, the contribution of examination and comparison will pay valuable dividends even if only demonstrating what not to do in software development.

The next step in the software development process, preliminary design, is the first phase in which actual coded modules can be examined for potential reuse. Applying reuse at this phase requires examination of those systems identified in the previous phases as having similar requirements, specifications, and performance characteristics. Careful scrutiny of selected target systems while simultaneously establishing the preliminary design should provide opportunity for the developer to compare and contrast actual modular breakdown with proposed modules. Because reuse in this phase entails identification and comparison of functions or hierarchical modules which can contain hundreds of lines of code, it is possible that actual code as-well-as module concepts or functions could be utilized in the developing system.

Reusing actual code will substantially reduce time spent on detailed design. It may be possible to incorporate directly (or after slight modification), any reusable modules identified during the last step. Direct injection of reusable modules will eliminate time spent on detailed design and during the coding phase. Additionally, this will also have positive impact on related testing, providing early information which could potentially impact other modules and associated testing.

If direct reuse of modules or hierarchical functions proves impractical, reuse can still be helpful in the

decomposition of the proposed system. Although not absolutely certain, the probability is high that new systems, unless incorporating radically new technology, will have some commonality of function or design with existing fielded systems, thus offering potential for reuse of decomposed forms.

During the coding and testing phase, the traditional concept of software reuse is applicable. That is, the traditional concept of line-by-line review of coded software, classified by standard domain analysis. This approach would be used for those modules which have not been the recipients of imported reusable modules identified in previous phases. Although it may sounder easier to commit these detailed modules to standard encoding procedures, the developer still runes the risk of generating errors in the code. And, as mentioned earlier, testing must be considered, developed, and tested. Testing of the new code can be time consuming. And while a line by line search (by domain) can be time consuming, it may still be quicker and cheaper than developing the individual coded lines. Reuse at this point also offers the potential of utilizing previously debugged and tested code, thus saving time. As sufficient code becomes catalogued, it may someday be possible to draw nearly 75 percent of new programs from reuse libraries, either as functional modules or as individual lines of code, with the remaining 25 percent

consisting of the requisite software bindings and new code to utilize technological advancements.

Although integration and system testing cannot directly benefit from reuse, the two areas will benefit indirectly. Because reuse can significantly reduce design and development times and thus the time spent on the associated testing and debugging cycle, the program should have more time in the overall schedule for integration and system testing. Additionally, the use of previously developed, and successfully tested and deployed software systems components can substantially reduce the integration debugging process time.

The maintenance phase of reuse candidate software programs should be referred to throughout the development process of new programs. Each step of the candidate programs for reuse should be analyzed for potential inclusion in the developing software program and then cross referenced with the maintenance documentation to determine faults or problems in the original programming. Any anomalies and errors detected and corrected in the original software and its test plan can then be applied or implemented into the new development. If correctly documented, the maintenance phase of programs targeted for reuse provides corrective guidance useful in cutting error detection time found in the original programming, and prevents repetition of errors found in reused software components.

Clearly, software reuse is applicable throughout the development process. It can be both cost and time effective, but there are limitations that should be readily apparent. Only successful, well-documented programs should be candidates for reuse. Programs which suffered from habitual teething problems during development should not be used, even if the program has been fielded. While a fielded program can be judged to be at least marginally successful, software which suffered through slow and error prone development generally suffers from poor planning, design, and execution, thus providing a poor model for new program development. As with any program development, a critical analysis should be performed of the risk involved with reuse candidate programs. Obviously, high risk programs based on dubious software programs should only be referred to for the lessons they can provide in error analysis and detection.

D. REUSE INHIBITORS

Although the software reuse procedure for new program development is fairly straight forward, there are a number of factors which have so far prevented employment of reuse on a widespread basis. These inhibitors cover a wide range of areas, but can be condensed into several primary categories. These categories cover 1) standards, 2) training and education, 3) management, 4) lack of centralized and cataloged assets, and 5) legal and contractual issues [Ref. 39:pp. 1-8].

While the management category would seem to encompass the reuse inhibitors from the program manager's perspective, in actuality, all of the categories bear some impact on reuse implementation. Each category contains inhibitors which actually affect areas outside what could be considered the bounds of that particular category. And, within each category, the separate inhibitors also carry overlapping influence.

The following is an analysis of the five primary categories of software reuse inhibitors. It will focus on the individual factors within each category and their impact on specific areas, applications, or implementation procedures and techniques of software reuse.

1. Standards

a. Lack of Standards

The lack of standards in such areas as hardware and software architectures, and commonly utilized software languages, perpetuated by what the DoD requests and requires and by what the industry and contractors provide, contributes greatly to the difficulty of attempting to implement software reuse. This is especially true of the military attempts at software reuse. Because the military uses civilian contractors to develop most software, the each contractor generally has commercial interests which produce software in a preferred commercially marketable language, there is a tendency by the contractors to develop DoD programs in the

same language of choice. Once the program is developed, it is then usually translated into the DoD preferred Ada computer language. (The term "usually translated" is used because the requirement for DoD to use Ada is fairly recent. A large amount of equipment utilizing a variety of software languages has been fielded prior to the implementation of this requirement, therefore it is impractical to expect this code to ever be compiled into Ada. A factor impacting the compilation of Ada in programs currently under development is the loose enforcement of the regulation. For any number of reasons, the software development contractor may be exempted from delivering a final software product in Ada.) However, because this is a higher order language, each line of which may be the transposition of several separate lines of another code which in turn can be composed of several levels of subcommands and routines, it does not decompose easily or simply when attempting to examine the code for potential reuse.

From the hardware perspective, the problem seems somewhat simpler to understand. Although two different programs may be written in the same language and may even have very similar applications or domains, they may be designed to operate in very different hardware architectures. The different hardware systems and their basic approach to program execution may effectively prohibit porting or reuse of other program segments onto targeted hardware. Whereas some

architectures rely on hardware solutions to specific problems, others rely on software solutions to address those same problems, consequently, the hardware design will dictate the software developer's approach to the program development.

b. Lack of standard or common development methodologies

Although the DoD is governed by a host of regulations designed to provide control and structure to the development process, civilian software developers are under no such regulation. Each software development company has its own internal program and guidelines. Consequently, the development process described earlier, which so readily lends itself to a structured building block approach and provides significant documentation so easily adaptable to reuse analysis, is not followed by those outside the DoD. As a result, adequate documentation for potential reuse may no be available. The usual drivers of poor or insufficient documentation is shoddy program design and incoherent structuring of modules, both of which dissuade reuse implementation. These programs often suffer a painful development process and are generally plagued with problems throughout their life cycle, making them unlikely candidates for reuse.

c. Lack of Common Notation for Describing Designs and Requirements

Although seemingly minor, this inhibitor to software reuse complements both the Lack of Standards and the

Lack of Standard or Common Software Development Methodologies. Just like spoken languages with their own unique alphabets, software languages have their own specific and distinct symbology and notation. A programmer experienced in one language may have only a vague understanding of another, rendering any attempt by the programmer to analyze a program targeted for reuse an exercise in futility. Although most symbology and notation has comparable representation in every language, the cost of translation and transposition can be prohibitively expensive for a project with such a dubious potential payoff.

Another problem occurs when attempting to measure the performance of programs against an accepted standard. The lack of commonly accepted hardware performance standards and software metrics prohibits the developer from effectively comparing the performance of one program or program segment against another. This is a very complex concept, as measuring performance is more than just operating against the clock. the measure of software performance must include allowances for hardware induced performance, as-well-as broad parameters which define singular computations and program executions. the measure of hardware performance on the other hand must take into account how each specific piece of software goes about the execution of its tasks, and negate those effects to accurately measure performance. The lack of standardized metrics or a defined process to test either hardware or

software components prevents any real attempt at even examining the broad base of existing software for potential candidate reuse programs.

d. Lack of Methodology for Extending Standards

There are literally hundreds of software development and consulting firms in operation in the U.S. today. Many of these firms are currently engaged in development or consulting efforts with the DoD, and are often in direct competition with each other. Consequently, there is seldom a free exchange of information between firms on progress in either software or hardware development. Additionally, each of these firms measures its progress against its own internally accepted standards, and while some of these standards may be shared or subscribed to by many firms, not all firms agree on all standards, nor are they necessarily the latest standards. Precisely because most of these firms are competitors, any advances in metrics, design, hardware, or development processes, are often kept secret by the developers t gain the most rom the advancement, either in monetary or technological terms. It is seldom in the firm's best interest to advance the general knowledge of its competitors. Current efforts by the industry to track progress and standards of performance consist of deriving information from trade publications, advertising copy, and reverse engineering efforts. consequently, there exists within the industry no mechanism or body of regulators (other

than the DoD) to decide what are the appropriate standards for the industry, how to ensure compliance with these standards, how and when to upgrade these standards, and finally, how to disseminate this information throughout the industry. As with the lack of standards, the lack of a mechanism to promote and disseminate those standards inhibits reuse by acting as a force multiplier for an already crippled industry.

e. Lack of Standardized Definitions of Reused, Common, Shared Software

In an industry bereft of standards, it is not surprising that a common definition for reused or shared software cannot be agreed upon. This is a simple situation of putting the cart before the horse. It would be impractical if not impossible to define such things as reused, common, or shared software without first addressing the industry wide problem of general software standards. Obviously, software reuse is inhibited by the lack of a standardized definition of what constitutes reuse. (If you can't describe it, you can't define it, and if you can't define it, you can't find it, and finally, if you can't find it, you can't use it!) Additionally, the lack of a regulatory body or dissemination mechanism adds to the inherent reuse inhibitions.

f. Lack of Well-Defined Reuse Methodologies

The compliment to the lack of adequate and standardized definitions for reuse or shared software is the lack of any well-defined or standardized reuse methodologies. While the industry cannot settle on standard software

development models, it would be totally unrealistic to believe the industry can settle on any standardized models for reuse implementation. Again, any attempt to implement an undefined concept across an entire industry can only meet with failure.

2. Training and Education

a. Reuse Inhibits Innovation and Reduces Competitive Advantage

Within the industry, software reuse is viewed as a rehashing of old ideas and technology. In an industry where there are literally scores of software producers, innovation is a market discriminator. A key strategy in marketing a software product is to differentiate the program from its competitors. This is usually done by focusing on some new and innovative feature or gimmick. Consequently, reusing previously released software eliminates this potential marketing approach. An additional factor that the software firm must consider is the actual software engineering workforce. If the aforementioned workforce views software reuse as a restraint on creativity or a hindrance to innovation, the company may be hard pressed to maintain experienced and talented software engineers. The perception that a firm cannot engage in software reuse while keeping talented, creative people on the payroll, and thus no produce innovative, cutting-edge competitive software products is a tremendous inhibitor to actual reuse application. This is true of the software firm, whether operating in the commercial market or the DoD contractor arena.

b. Lack of Readily Accessible Information on Reuse

Although the concept of software reuse has been around for years, there is a dearth of information readily available on the subject. General industry attitude combined with the lack of standards and methodology has left software reuse in its infancy while the primary focus of the industry promoted evolution of program engineering.

Software reuse information is skewed by rumors and falsehoods about the subject. This is the result of the ill informed or misinformed generally interjecting their bias into the available information. It is important to remember that programmers are often paid by the number of lines of code they generate, and consequently find it in their best interest to inhibit something like reuse which they might perceive as a threat to their livelihood.

This overall lack of quality information on reuse has stunted interest in the subject and the proliferation of usable information. The same programmers who may see reuse as a potential financial impingement are also the same programmers that write industry magazines. It is entirely possible that information about reuse could be stifled for reasons of self interest. Without accurate and adequate information or any mechanism to spread that information, it is doubtful that reuse will ever be implemented on a wide spread basis.

c. Limited Training for Reuse

For those interested in reuse, whether they are managers or engineers, contractors or the Government, there is little or no training available. As mentioned earlier, there is a dearth of standards within the industry. Consequently, very few companies or organizations are inclined to invest money, time, and resources into training personnel to engage in or manage reuse of software components. The lack of training also impacts the amount of information available on reuse. There is a dependency cycle in which information is needed to inaugurate training, but training is needed to develop information. Once the cycle begins, it will be self-sustaining. However, getting the cycle started may be a monumental task.

d. Lack of Knowledge and Training of Data Rights and Licensing Procedures

Although this might be considered as a topic under the Lack of Available Reuse Information category, it is really more of a legal problem than an information problem. Proprietary rights and data rights of published or contracted software are by law the property of the developer (except where contracted developers give up those rights as part of the development project) and can be used only under license rom that development firm. For any potential reuser, there must be a license agreement, not just to use the software, but possibly to decompose it, alter it, and finally combine it

with other code from similar sources. There are a number of problems with this concept.

First, to submit a program or programs to reuse will require substantial documentation of the software and testing. The producer, in order to protect his interests in this process will need to engage in management of the software which, for an independent producer especially, can take considerable time. Either the producer manages the program himself or the firm establishes an internal mechanism to manage this process. This would require manpower, facilities, equipment, and cash. Obviously, this investment must be weighed against the potential profit to be made through licensing reusable software. An additional problem is potential liability for software problems which may come from a firm's reused software. In an era of intense litigation over producer liability it could be catastrophic for the developer if his software caused a major software crash for another developer.

For the potential reuser, the problems are even more complex. The potential reuser must either be ready to spend great amounts of time and money to analyze potential reuse candidates, or he must develop a mechanism to conduct reuse business. Although this sounds relatively simple, the establishment of a reuse management mechanism for the potential reuser would be costly and have a tendency to grow. The essential structure would need to include a manager

steeped in software engineering as-well-as contracting. A bevy of software engineers familiar with internal projects requiring reusable components would be required to research and analyze each and every potential reuse candidate program. The section would require both lawyers and contracting personnel to work out the details of agreements which would allow the software engineers to examine other programs as-well-as use favorable components. And finally, any effort of this type will require a multitude of administrative people to manage the records and documentation.

A final problem which may plague the potential reuser is the fact that many software forms are here today and gone tomorrow. Although the programs are copyrighted, the company may no longer be in business. Any reuser is still bound by law to license the software for reuse. This means that the reuser must find an organization or person with proprietary rights over the software. This can be a long and tedious process and may not be worth the time and effort to conduct a search versus just developing the software from scratch.

Until the legal fundamentals are worked out and guidelines established and firms are ready to make a concerted effort to implement software reuse, this area will continue to be a problem and will definitely impact reuse implementation.

e. Software Common Practice of Redesign/Redevelop Versus Hardware Incremental Development Practice

Within the automation industry, there are two distinct practices with respect to hardware and software development and improvement. Firms generally approach hardware upgrades or revisions using the incremental improvement technique. Their approach to software on the other hand, is one of new program development instead of measured improvement.

Essentially, firms utilize existing hardware platforms and apply focused technological improvements to specific areas of the platform, incrementally improving performance. There are several reasons for this hardware improvement approach. First, the pace of hardware improvement moves only as fast as technological advancement. Although revolutionary improvements do happen within the industry, most of the effort is focused on improving existing technology. Consequently, great technological strides or revolutionary improvements are few and far between. Second, is the matter of economics. All of the firms in the industry are in business to make a profit, and each firm does this in a variety of different ways. Firms make money on providing upgrade components to existing equipment -- again, an effort to capitalize on technological advancement. Another method for gleaning a profit commonly utilized by the industry is to offer a wide variety of models utilizing common components or technology similar to practices found in the automobile

industry. Finally, and most controversial, is the practice of utilizing planned technological obsolescence and controlled technological insertion. This method calls for utilizing combinations of components which have a limited or planned life-span with respect to leading-edge technology. This planned obsolescence is coupled with carefully calculated technological insertion. Essentially, a firm will time the release of technological improvements as a marketing tool to boost sales where current machines offered for sale have lost their technological edge to the competition. This idea ties in to the concept of maintaining a desired level of market share. If a firm wished to maintain its current level of market share, the firm may hold some improvements in reserve to counter competitive efforts by the competition to increase market share. And finally, incremental improvements are the backbone of the lucrative upgrade market mentioned earlier.

Software development on the other hand is viewed as a cottage industry within the automation field. Because software development is generally less equipment and personnel intensive, it is viewed as being easier than hardware development. This view is predicated on the industry's lack of standardization with respect to almost every area of software development. Instead of teams of engineers working together to develop improved hardware, the software environment is populated by individuals, meticulously and painstakingly developing and testing a program on the targeted machine. The

industry views software development more as an art form than a science. Consequently, software developers are generally subject to less management and control than hardware developers. This lack of tight control eliminates any incentive within the firm to utilize software reuse. After all, creativity is viewed as an asset in the software field and reuse is held to be in direct contradiction to that idea. And much like artists, software developers reflect these values and beliefs, consequently, left to their own devices, few opt to review old programs for usable parts or pieces. Finally, because there are considerably fewer resources required or utilized in software development than hardware development, it is viewed as being considerably easier than hardware development.

It is necessary to make a clarification at this point. Software is commonly released in versions, with each version representing an improvement over previous versions. These improvements are usually nothing more than refined or debugged previous editions of current software. Therefore, these new versions of the software are essentially not true revisions or design changes of the programming. Eliminating version revisions as true improvements, actual software improvement comes in the form of new programs, offering new capabilities, tools, and quicker program execution.

The two different approaches stem from the manner of development for each of these areas. Hardware development

is expensive due to its nature. Development or improvement of hardware products requires expensive laboratories, equipped with state-of-the-art test, diagnostic, and measurement equipment, capital intensive production facilities, expensive distribution networks, and extensive training for maintenance personnel. Hardware is the product of teams of tightly managed engineers operating in a structured environment. Improvements in hardware are incremental or marginally evolutionary versus revolutionary. Firms have found it in their best interest to tightly manage these assets to appreciate the highest possible return on the dollar. Whereas software improvements seem to reflect flashes of individual brilliance, unencumbered by management or large quantities of equipment. Management is less apt to indulge itself in an area that defies standard organizational structure, management techniques, and time lines. Until the industry institutes software standards, individualism at the expense of reuse will remain the norm.

3. Management

a. Lack of Program Office Incentive to Initiate Reuse

Without exception, software development within the DoD has been focused at the individual program level as opposed to a broad-based focus aimed at multiple reusable applications. Because each Program Manager is tasked with the development of his particular program and will be judged accordingly, there is little interest on the part of the PM to

go beyond the program mandate. Even with programs that must be tied together, such as the Army Tactical Command and Control System (ATCCS), which requires the interface of the separate software development programs of the five BFAs, there has been no effort to exploit software reuse, structure standardization, or interface bindings.

The PM's area of responsibility, which can include both hardware and software development, really only encompasses a small domain with respect to either of these areas. Consequently, the PM has only a limited ability to influence anything outside of his mandated area of responsibility. Coupled with this limited ability to influence outside areas, is the political danger to the PM of expanding his control or influence into another PM's domain or program.

Although the Army presents itself as an apolitical organization, which is true of the tactical and operational portions of the force, it is not necessarily representative of the acquisition and procurement areas. These areas are structured along the lines of civilian organizations and are involved in similar pursuits. The program development and acquisition field is mostly the domain o the Army's civilian workforce, and very much emulates the politics the civilian industry it mirrors. Domains of influence and spans of managerial control within the Acquisition Corps are often jealously guarded, with interlopers being shunted, ostracized, or victims of political paternalism.

b. Lack of Personal Recognition or Economic Incentive for Developer of Reused Components

Putting all the factors together, it is obvious that there is no incentive for the individual developer to either develop or utilize reusable software. The developer is generally paid on a by-line production basis, consequently, to produce reusable code or implement such code would be tantamount to reducing or eliminating one's livelihood.

Additionally, software development management is generally pulled "from the ranks," perpetuating the relaxed, almost loose management atmosphere prevalent in most development companies. Because of the relatively unregulated development atmosphere, there is little guidance or direction aimed at reuse employment or development.

As mentioned earlier, the lack of industry standards and formal mechanisms to either disseminate information or govern rights of reusable software serves again to inhibit the individual software developer from either utilizing or developing reusable code. Because code and documentation are not readily available without extensive legal negotiations and because there is no royalty mechanism in place to reward the developer for his efforts, there is no incentive to move in this direction, especially for the individual.

c. Lack of Trade-Off Mechanism Between Requirements and Reuse

A requirement is by definition, a need. Software is written to satisfy the need. Any differentiation between the requirement and the performance of the software does not qualify as meeting the need. In order to implement reusable software, the requirements must be reasonably flexible or generic in nature. Unfortunately, however, software requirements are not usually flexible or generic. Consequently, it is difficult to find the necessary middle ground to satisfy the demands of both.

The problem stems from the development process. The requirements for any project are drawn up early on and are the result of mission need statements generated from the user community. Because these needs are drawn up without regard to software development or software reuse, no compromises or trade-offs are established. Consequently, there is little room for software reuse if the requirements are to be effectively satisfied.

d. Lack of Reuse Cost Models or Metrics

As stated earlier, an industry wide lack of standards in software development, architectures, and metrics has effectively deterred the development of any reasonably reliable cost models for software reuse. This is almost the proverbial chicken and egg situation. In order to determine the cost effectiveness of implementing software reuse, it is necessary to evaluate existing software reuse cost models.

However, without effective and widespread utilization of reuse, valid cost models cannot be developed. As with any new technology, it often requires investment of a great deal of time, money, and resources to begin the initial venture. Only after relatively large and risky expenditures of capital does a company normally begin to see positive returns on investment. The current state of the software reuse industry is much the same way. The current measure of risk has so far inhibited reuse and the associated models which could someday prove its profitability.

e. Limited Vision or Leadership for Reuse

As stated earlier, management of software development is very much an inside job, consequently, those boosted to leadership or management positions are often interested in maintaining the status quo versus implementation of new cutting edge ideas. This is not necessarily because those elevated to management are against new ideas, but more because of the reputation established by the company before the new leadership took over. Essentially, some companies are known for certain software traits, designs, or architectures which are accepted and expected within the industry. To break with this established convention can be costly in terms of lost customers.

However, the greatest inhibition comes from a general lack of knowledge about reuse in general. For the manager, the requisite questions of how to classify software,

where to find the necessary reusable software candidates, how to navigate the legal obstacles to reuse, and how to motivate the actual developers to implement reuse are insurmountable simply because of the relative infancy of the field and the lack of managerial experience or established guidelines in this area. There is one other simple, yet looming reason for the lack of reuse implementation by management. A great number of those rising to managerial positions in the software development field do so by default. Most lack the drive and aspirations found with managers in other areas of business. Consequently, there is a marked reluctance to aggressively pursue such controversial and dubious endeavors as software reuse.

f. Lack of Knowledge and Training on Data Rights and Licensing Procedures

Software, like nearly every other product on the commercial market is surrounded and supported by a host of laws designed to protect the producer's product, his ideas, or development process from being copied without permission or monetary compensation. Copyright laws similar to those covering audio and video tapes and discs govern the software industry. However, unlike audio and video tapes, software, although relatively easy to copy or pirate, is of little use without documentation, and of no use unless it can be decomposed. Therefore the problem here is not piracy, but licensing -- the authorized use of all or a portion of a piece of software (to include documentation) by another company in

exchange for monetary compensation. This is a relatively new field with respect to software, and lacks precedents for establishment of guidelines or rules.

Because of the vagaries of software development and business, questions and concerns addressing the potential profitability of a program which contains reused software, potential liability of the owner of reused code, and potential licensing of software components which are composed of new code as well as reused code pose special problems for software development companies. Although many companies are faced with similar problems which impact the decision making process, very few face the situation wherein their product can have an indeterminate effect when imported into another program. The potential outcome of such a situation could be devastating if for some reason the reuse software creates problems or systems failure. The issue of who is responsible, the importer or the original developer, is very much in question in such cases and has yet to be determined in potential licensing agreements.

The issue of product liability in cases of software failure is only one problem however. Just as important, at least to those individuals who develop software is the issue of by-line payments. Should the individual developer be paid for the reuse of his product, and if so, how much, and how should this issue be approached for a software program which contains reused software and is itself a candidate for software reuse?

This may be moot if the target software is altered to facilitate reuse. It does however, raise another issue -- that of technical propriety. If a piece of reuse software is altered to facilitate reuse, does the original developer remain liable for problems? Who controls licensing of altered reusable code?

Finally, when the Government engages a contractor in software development, the Government generally requires the proprietary rights with the software. This presents problems when the software contains reused code. What are the legal rights and obligations of the original developer of the code, and what are the rights and obligations of the second developer or reuser with respect to the different parts of the code? Who is responsible for the performance of the code, and who takes responsibility for failure?

These and other issues have yet to be addressed by either the Government or the software industry. Until these questions are answered however, potential legal obstacles will continue to be major obstacles in implementing software reuse.

g. Contractors Not Paid for Productivity

It should be clear by now that the bulk of inhibitors work in concert to prevent the widespread implementation of software reuse. But regardless of the wide range of reasons for lack of implementation, the bottom line in most cases is money. Today, contractors are not required by the Government to utilize reuse. And because of all the reasons

stated above as-well-as the implications for the industry -- widespread implementation of reuse would substantially reduce the number of competitors in the business -- there has not been a rush to reusable software. Further, it would be ludicrous to assume that any developer would either design a program to be reusable or utilize reusable code without some sort of incentive, either in terms of more follow-up contracts or direct monetary compensation, while at the same time facing possible elimination from the industry by the very thing under development. An developer interested in perpetuating his business realizes that reuse could be a serious threat to continued operation. As pointed out earlier, software developers view reuse with some skepticism, realizing that widespread implementation could change the face of the industry, eliminating some of the players and the way business is done. For any concept with such a potentially catastrophic impact, both the short and long term monetary rewards must be enormous. The current system falls far short of offering this type of reward.

h. Other Potential Reasons

Finally, there are a number of obvious inhibitors that should be mentioned. These inhibitors need little or no explanation, and are listed as follows:

(1) Budget and schedule pressure.

(2) Increased organizational interdependence due to reuse.

(3) Redesign versus redevelop mentality of program offices.

(4) Profit and greed on the part of the developer.

(5) Software is not viewed as an asset in program development.

4. Lack of Centralized Catalog of Assets

a. Too Few Libraries

The current approach to utilizing software reuse is to catalog lines of code and provide access to that code through a system of libraries. These libraries would categorize code by a multitude of characteristics and provide the code and its relevant documentation through an automated search and retrieval system. Currently however, there are only a few woefully small libraries currently in existence.

Although the library concept is the most reasonable approach to real world implementation of software reuse, it also presents a number of unique problems. First, who is the proprietor of the library system -- the Government, commercial business concerns, or some non-profit organization? Second, what categories are logical and reasonable for the classification of the very broad spectrum of software? Next, with such a prolific amount of software already in existence and more being produced every day, how many of these facilities will be enough to meet requirements? And finally, who should pay for the initial capital investment necessary to establish a series of libraries and their requisite automation links? These issues will be some of the first and most important to be addressed once the Government and industry

begin to accept and develop software reuse on an wide spread basis.

b. No Easy Way to Search For or Retrieve Components

Touched on earlier, reusable code will most likely be made accessible through a series of libraries. Although this should make the job of locating large quantities of reusable software components easier, the task of finding the right components within this large reservoir of software can be monumental. A major inhibition to reuse is the time required to locate potentially reusable software that conforms to the needed template. Even with automated libraries, the shear volume of potentially reusable code could literally take weeks or months to comb. It can take twice that much time to test.

Another problem is how to search for the necessary code. There are millions of potential categories or software classification, ranging from overall program classification to subroutine and individual code segment classification. Most of these components fall within the domain of several of these potential categories, making the task of assigning code to a specific category even more difficult. Any programmer seeking to utilize reusable code will need to have a detailed description of the type of code required. The process could be equated to trying to find one specific piece of an unassembled jigsaw puzzle. Until a coherent and easily utilized search

and retrieval system can be established and implemented, software reuse cannot be a viable development tool.

c. No Defined Way to Test or Accept Components

Having examined the classification and storage problems as-well-as the search and retrieval problems, the next logical step is to examine testing and acceptance criteria. Once a developer has waded through the system to locate reuse candidate software, it must be tested to ensure the code fits the need or requirement. Currently however, the only test available is to actually integrate the reusable code into the program and test for functionality. The testing process can be expensive and time consuming. Of great concern to any developer is the quality of potentially reusable software. As with any endeavor, there is a right way and a wrong way to do things. The same is true for software. Shoddy or unproven techniques used to develop software make for an equally shoddy or unreliable product. Poor quality software can make integration difficult or impossible. It can also lead to difficulties when trying to layer or host application software on top of an operating system utilizing reused software, or vice-versa.

But of even greater concern are the proprietary rights issues described earlier. Should the software prove unusable, is the developer still obligated to pay user and licensing fees? Further, does the library serve as the licensing agent or should each interested party engage in

negotiation to arrive at some mutually agreeable arrangement? And in reference to the quality issue, what is the developer's recourse after discovering he has inadvertently reused a poor quality piece of software? These and other questions must be answered through the legal system, library system, and evolving efforts to standardize the industry before they can stifle software reuse in its infancy.

d. Configuration Management Across the Library Network

Software products generally go through an upgrade/ update process that works to create newer and more refined versions of a particular program. Often these changes or upgrades are so prolific and frequent that five or more versions of the software can be in use simultaneously. The latest version is usually the most refined and error free of all the configurations available. Keeping up with these changes and associated documentation means replacing current versions contained in the libraries with the updated versions. This can be a costly and time consuming process, and leads to more questions. Specifically, should upgraded versions of software be provided to developers currently using earlier versions of the software? Are amended licensing agreements necessary? Clearly, as with the other categories of inhibitors, a great deal of work will be necessary to overcome both the obvious and subtle problems in instituting reuse on an industry wide basis.

5. Legal and Contractual Issues

Most of the major legal and contractual issues concerning the implementation of software reuse have been mentioned above. They are complex issues with far ranging implication. Addressing these problems through the legal system will take years, and in some cases will result in dubious outcomes. The complexity, cost, and extent of these issues, possibly more than any others, will prevent the software industry from accepting and adopting reuse. Few companies today can stand the withering legal assault that marks a product liability suit, nor can most companies accept the staggering fines imposed for copyright infringement or piracy of proprietary data. Neither do most companies believe it is in their best interest (profitable) to be obligated to maintain possibly legally mandated technological upgrades of software in libraries, insure reusers receive updated or upgraded software and appropriate documentation, or maintain large staffs of legal personnel to look after the company's best interest with respect to reuse. Therefore, most companies have so far given reuse a wide berth. And, because of the drawn out legal process and exorbitant expense of retaining the requisite legal expertise to address these issues, firms will maintain that distance for the foreseeable future.

E. SUMMARY

Obviously, the software development cycle is a complex process, involving risk, time, money, equipment and personnel. Yet even more complex is the reuse equation of software development. The PM can have either a positive or negative effect on software development, even given a lock-step software development process. As nearly every step, the PM or those working for him have the ability to develop code and implement measures to aid or retard the ability of the software to be reused.

For every advantage demonstrated by the implementation of software reuse, there seems to be a myriad of reuse inhibitors. These inhibitors all fall into five specific categories -- Standards, Training and Education, Management, Lack of a Centralized Asset Catalog, and Legal and Contractual Issues. And while individually most of these obstacles, whatever the category, are small, taken as a group they are most daunting. Unfortunately, there is no way to separate most of these inhibitors from each other. Most are directly related to others, both within and across categories, often instilling a sense of dread and hopelessness on would-be reusers.

While this chapter has focused on the development process and the associated roadblocks to implementing software reuse, the primary driver or force involved in the development process for the Government is the Program Manager. The human

factor can be the greatest asset in implementing reuse, or it can be the greatest adversary against implementing reuse. This is the built-in bias of every software development program. It is that portion of the program that can be both the easiest and the hardest to change.

IV. THE VIEW FROM THE TOP

A. INTRODUCTION

The documented inhibitors to software reuse described in the last chapter clearly demonstrated the propensity of management to fail in the decision making process as opposed to succumbing to technical shortcomings. This is simply because most of the technical inhibitors to reuse are actual, identifiable problems which will have real answers or solutions. Most of these problems are being addressed and are in the process of being resolved.

The biggest obstacle to implementation of software reuse seems to be breaching the nearly impenetrable wall of human nature and resistance to change. Specifically, the beliefs, techniques, and bias produced by years of involvement in program development stand as staunch testimony to the human aversion to trying new things. Once someone learns what works with respect to any given area of application, human nature resists the urge to exchange that "proven" approach for an unproven or untested method. By the time a person reaches the position of program or project manager, he has accumulated years of experience in program development. Unfortunately, the greatest lesson learned seems to be that the penalty for failure can be an abbreviated career. Consequently, the potential for inclusion of a relatively unproven process such

as software reuse into a software development project by the PM would appear to be inversely proportional to his career aspirations. In order to either confirm or discount this idea, at least with respect to software reuse, numerous interviews were conducted with program, project, and product managers as-well-as their deputy PMs, and hardware and software division or directorate chiefs.[8] What follows is a discussion of the personal views of those most involved and concerned with software development within DoD.

B. PROGRAM/PROJECT/PRODUCT MANAGER

More than any other individual, the PM has the ability to have the greatest impact on the development of any project. The PM is ultimately responsible for program development and necessarily has control over the assets within his organization which have direct influence on the final product. The PMs interviewed were involved in the development of either systems utilizing embedded software or separate software systems. Most, but not all, were engineers, with electrical engineering being by far the most predominate discipline. Two of three PMs were senior U.S. Army field grade officers, with the other third being senior civilian Government employees. All were at least conversant in both hardware and software systems technology. However, as might be expected, PMs

[8]Interviewees were afforded anonymity, hence no names, ranks, or program/project/product titles are given. The data presented here are by necessity a composite picture of the results of over thirty interviews.

responsible for embedded systems were generally more knowledgeable about hardware than software. The PMs with non-technical backgrounds were less focused on the specifics of either hardware or software development and were much more concerned with budget and schedule problems. All of the PMs interviewed took control of a program already in progress. Finally, all of the programs examined had both schedule and budget deviations from the program baselines.

Regardless of the type of software system being developed -- embedded, operating, or applications -- there are some common factors with respect to software that apply to both types of programs and affect all PMs. Although all PMs interviewed were aware of the concept of software reuse, most had little in-depth knowledge of particulars of the subject. Few could either define reuse as envisioned by DoD, or describe the necessary application environment. Fewer still believed that their particular program could benefit from reuse. Most of the PMs were skeptical over the viability of their program to even be considered for reuse, either as a recipient of reused code or as a potential candidate for reuse in other programs. All of the PMs cited some distrust of other PMs' software programs with respect to application within their own domain. (It is interesting to note that

while being members of the Council of Colonels[9], many of those interviewed had only cursory knowledge of the other programs' particulars.)[10] Additionally, most voiced concerns about responsibility for system failure when importing or exporting code. No PM was willing to risk further complicating his program with software reuse, citing potential problems such as the time, money, and resources necessary to search for a candidate code. Nearly all expressed concerns about time necessary to integrate reusable code into the developing program, and the potential for schedule slippage if the testing demonstrated problems with the imported code.

All the PMs agreed that there were significant differences between programs employing embedded systems software and those utilizing operating and applications software programs. All agreed that embedded systems were by far easier to control in terms of software development, with most conceding that control over hardware development augmented software development. The PMs responsible for embedded systems were able to control both hardware and software development to be

---

[9]The Council of Colonels is an advisory and steering committee made up of military O6 (or civilian equivalent) PMs within a PEO. They deconflict and coordinate potential high-level problems between related programs within a PEO which cannot be settled without an executive decision.

[10]Once the initial interview was conducted and a baseline of current knowledge established, current trends and advances in reuse as-well-as current and anticipated efforts by the DoD were described and explained to those interviewed. Additional questions concerning the potential impact of reuse on programs were asked once the current reuse efforts were explained.

mutually supportive in the development process. Those PMs working on embedded systems also saw greater potential for reuse in their programs once the concept was explained. Specifically, they believed that subsequent upgrades to their particular system could readily utilize reuse, baselining from the original code, provided the intended upgrade was compatible with current hardware and software architectures. But again, all hedged when asked if they would willingly import code from external sources into their programs. No single PM was willing to accept the risk of utilizing reuse in their program development, even when potential sources of reusable code, similar domain architectures, and proven techniques were readily available.

PMs responsible for the development of operating or application software were even less receptive to the idea of software reuse, at least from the importation point of view. without question, all of the PMs in this category controlled programs much larger and more complex than those involved in embedded systems. Not surprisingly, the viewpoint that each had the most difficult and complex program was prevalent among all the PMs responsible for this type of development. All of the PMs in this category held the attitude that if their program did not develop the software, it could not be trusted to meet their requirements. Additionally, few of the PMs recognized any similarity between their programs and potential reuse candidate programs. Although most PMs had no problem

with exporting their software to other programs, none of the PMs were willing to take responsibility or system failures attributable to their exported software. All the PMs believed the burden of responsibility for selecting proper reusable program segments and efforts to integrate selected reusable software components rested with the gaining program PM.

All of the queried PMs stated unequivocally that initiating software reuse was essentially outside of the PM's charter, and should be considered exclusively the domain of the Program Executive Office. The consensus was nearly unanimous that the resources did not exist at the PM level to keep up with current trends and available reuse technology. Additionally, the PMs pointed out that they essentially operate in a vacuum, with only limited access to information concerning software development in other programs. Instead, they emphasized the PEO's responsibility for overseeing and integrating the efforts of PMs.

C. DEPUTY PROGRAM/PROJECT/PRODUCT MANAGERS

By far the least vocal and least cooperative of those interviewed were deputy PMs. Most, harboring further career aspirations, were reluctant to comment one way or another on the potential for implementation of software reuse in their particular programs. A few however, were responsive to questioning. All of the DPMs interviewed were Government Civil Service employees, and possessed technical backgrounds about evenly split between electrical engineering and

software/hardware engineering. Of those who cooperated, most were aware of the concept of software reuse. The DPMs were, by far, more knowledgeable with respect to reuse than anyone else in the program, including the PMs. In fact, those responding to the interviews were either directly or indirectly involved in the DoD and DA efforts to initiate software reuse on a widespread basis. Although many of the DPMs were more familiar with reuse than the PMs they served, they mirrored the general attitude of the PMs with respect to reuse implementation. Part of this can be attributed to the politics of survival in Government service. However, part of the responses posted by DPMs can be attributed to their greater depth of knowledge about reuse and its potential benefits and pitfalls.

While the PMs recognized a significant difference in embedded versus application software systems development, the DPMs generally discounted this perception. Nearly all the DPMs thought that application of reuse was more a matter of correctly defining requirements and domains than a systemic characteristic of their particular type of program. Although most conceded that the embedded type systems could potentially be easier targets for applying reuse, all agreed that reuse was equally applicable to software applications programs.

The responses of the DPMs, reflecting a substantially different outlook on the potential of reuse, fives a clue to the ultimate potential of software reuse. Of those DPMs

responding to questions, nearly all felt that practical, simplified software reuse application would be a reality in the near future. Most felt that reuse was in its infancy, with incredible potential for application across almost every future software development program in the U.S. Army. They also agreed however, that to be effective, reuse would need to be expanded to include development programs within each of the other military Services and civilian agencies. This type of response reflected a far greater understanding of the potential for reuse than that represented by the PMs. Although all of the DPMs expressed concerns similar to the PMs about the resources needed to locate, integrate, and test potential reusable code, all of the DPMs dismissed the problems, expressing the belief that further development and maturity of the reuse effort would eliminate these problems over time.

Of particular interest were DPM comments about their PMs' lack of understanding of program particulars involving aspects of software development. Most were critical of what they described as the PM's under control of the development process. Specifically, a couple of DPMs believed that the PM assumed that the software development cycle was the <u>easy</u> part of the program and concentrated more on hardware development than software development. Consequently, their programs suffered moderate to severe teething problems with software development costs and schedule. However, this can possibly be

attributed to human nature and one's belief that their way is always the better way.

As with the PMs, the DPMs believed that initiation of software reuse was outside the domain of the PM. All felt it was the exclusive domain of the PEO. Citing the PEO's broad span of control and influence. Additionally, most of the DPMs felt that the PEOs should develop and maintain the mechanism (division or directorate) to identify potentially reusable software components for import into programs and export out of programs.

D. HARDWARE/SOFTWARE DIVISION CHIEFS

Of all those interviewed, the division chiefs (DCs) were the most vocal and passionate concerning software reuse at the PM level. All of the division chiefs interviewed were Civil Service employees with either electrical engineering or software engineering backgrounds. Most had been with their particular program since its start, and all had seen PMs come and go. The mix of programs was about evenly split between those developing embedded software systems, software only, and joint hardware/software systems.

All of the DCs were familiar with the concept of software reuse. However, only about half had a good understanding of the mechanics of reuse or its potential. While most tried hard to stay current in the software field (regardless of program type), most found that the demands of the job often overrode these efforts. Even so, over half felt they were, by

far, much more knowledgeable than the PM about the mechanics of the software development process, and about software in general. Nearly all felt the PM exercised too much control over the everyday operations of their divisions and that the PM's lack of first hand or current knowledge of the latest trends in both the software and hardware communities contributed to problems with the program development. Essentially, nearly all the software DCs felt that they were the in-house experts on software and that their talents were not properly utilized. Additionally, most of the DCs were openly contemptuous of outside contractors utilized by the PM to augment either hardware or software development. Many expressed frustration and felt they were constantly being "shown up" by the lack of coordinated effort within the program. The main complaint was that outside contractors were utilized to pursue specific ideas without the coordination of the critical program development divisions.

With respect to the implementation of reuse to their specific program, by a wide margin, the DCs felt that reuse would place an undue burden on their divisions within dubious potential for payoff. Most felt reuse would be an added burden in terms of time and manpower, and none felt that their division would benefit by expanding the number of personnel within the division. Only one had any real knowledge of the DA and DoD efforts to develop and institute reuse into the development cycle, but most distrusted higher levels of

management to effectively develop and implement a broad based software reuse program that might ease the PM level burden. Software DCs presented a constant picture of frustration and looked upon reuse in a decidedly negative light, counting it as just one more problem to overcome.

Because of the potential impact of software reuse on hardware architectures, hardware development division chiefs were also interviewed. While most has some experience with software, few were really current in the field, and none had any knowledge of current reuse efforts. Once the program was explained to the hardware DCs, the most common reaction was for the DC to point out all the potential hardware development problems implied by reuse. Essentially, all believed that initiation of reuse would severely limit options available to hardware developers. All of the hardware DCs felt that software would become the dictation precedent with respect to any development activities, whether upgrades or new product developments. All of the DCs expressed dismay that hardware development would be significantly retarded in order to accommodate reusable software, and that hardware would take a backseat to software. This obviously reflects a fear of loosing substantial influence within the program development domain.

The attitude that hardware is the program driver or central focus of program development was confirmed in the discussions with both the hardware and software division

chiefs. While this is only relevant in embedded systems and programs developing both hardware and software, it is the predominate attitude in these programs. the hardware DCs clearly understood they had more influence over the direction of the program, while the software DCs clearly felt they were often the victims of the hardware developers.

E. FINAL OBSERVATIONS

While most of those interviewed had hard and fast opinions either supporting or condemning software reuse, all were professional in their approach to their work. While some may have had reservations about the implementation of reuse into the development cycle, all admitted they would support whatever program was put before them. All experienced frustration at attempts to maintain currency with respect to technological developments in their given field, whether hardware or software development. And finally, nearly all felt the next higher level of management held the key to successful software reuse implementation, while simultaneously condemning that very level of management for current program shortcomings and problems.

As for the PMs, they held the unanimous opinion that any PM selected for an automation program, whether hardware or software, needed a technical background either in electrical engineering, computer engineering, or at the very least, computer science. Most felt that taking over a program already under development as opposed to start-up, locked the

PM into an unalterable course of action. the consensus was that once the program reached a certain level of effort, any opportunity to altar the basic direction of the program was lost. For example, if a software development program was experiencing difficulty after spending significant amounts of money, little recourse was available to the PM except to spend even more money to fix the problems. Starting over or scrapping a portion of the program were not considered to be viable options by the PMs. And finally, without exception, all of those interviewed felt that for any software reuse program to be successful and effective, it would need to be top driven, administered at DA or DoD level, and controlled locally at the PEO level.

F. SUMMARY

As mentioned in the introduction to this chapter, human nature and resistance to change form an nearly impenetrable wall that may well be considered a tangible reuse inhibitor within the program office (in the same sense that the inhibitors described in Chapter III are tangible). The case could be made that this is essentially another category of inhibitor, populated by the collective fears, doubts, mistrust, and abject pessimism of those tasked to implement software reuse at the program management level. This was readily demonstrated by the numerous negative responses elicited from program management personnel with respect to implementation of reuse into their programs.

Regardless of rank or position within the program structure, all those interviewed believed that the PM had the greatest potential impact (with respect to implementation of software reuse) on software development, but was the least technically competent to affect critical program changes. Everyone interviewed also believed that software reuse would increase the already overwhelming workload without increasing the available program manpower. The bottom line, repeated at every level of management, was that software reuse will substantially increase program risk. Without exception, all those interviewed were admittedly risk averse. Consequently, it comes as no surprise that there is a stone wall of human emotion which must be breached before software reuse can become a reality.

To effectively overcome this obstacle, it will be necessary for the DoD and DA to address the previously categorized reuse inhibitors and barriers. Admittedly, elimination of these inhibitors alone will not automatically open the doors to reuse to reuse acceptance, however, it will serve to augment the intense education effort that must be implemented to overcome the human aspect of the problem. Because these inhibitors form the foundation of the human trepidation about reuse, efforts to overcome these issues will also work to break down the human resistance to change. Therefore, the key to initiation of the reuse effort must lie in solving the problems described in Chapter III.

V. ADDRESSING THE PROBLEMS

A. INTRODUCTION

The previous chapters have examined the systemic problems in implementing software reuse into the program software development process and the effectiveness of the Program Manager in influencing the infusion or omission of reuse in software development. Although these problems are wide ranging and seemingly all encompassing, they are not insurmountable. This chapter will attempt to address some of these problems and inhibitors with potential solutions. First, possible solutions to some of the systemic inhibitors presented in Chapter III are examined.

B. SOLUTIONS

1. Standards

a. The DoD should sponsor an effort to standardize software requirements, metrics, notation, and design methodologies. This would serve a two-fold purpose. First, standardization within the DoD would reduce program development cost. Cost savings would be realized not only through the potential implementation of software reuse, but through the reduction of effort involved in determining program requirements, measuring and comparing software effectiveness, and developing the required software through the use of standardized methodologies. Second, standardization will

reduce the time necessary for development of software components. Standardization of development methodologies, especially those which can be integrated with or imposed on contractors' current methods will streamline the software development process and allow a greater degree of control to be exercised by the PM. Implementation of reuse, made possible through widespread standardization, will enhance the process and further reduce development times by allowing earlier integration start times. The PM's control of the program will be strengthened by allowing him to refer to previously completed programs to foresee potential problems and solutions as-well-as established time lines to measure program schedule effectiveness. And, in a situation where program development time is tied directly to cost, shorter development time and fewer delays will further reduce expenditures. Additional benefits which may be realized through standardization would be the potential for more competitors for DoD software development contracts, should the DoD standard also receive widespread acceptance as the commercial standard. Such a proliferation of standardized methodologies, notations, and metrics could conceivably increase contractor competition, further reducing costs and improving the final product.

Although the initial investment in this effort would be expensive, the potential returns are immeasurable. As a practical approach, standardization should probably be

implemented through IEEE or ISO standards/specifications. [Ref. 38:p. 21]

b. Small teams should be formed to develop initial architecture and interface standards/specifications for specific domains. These teams should also establish methods for updating and disseminating standards more quickly and efficiently. Standardization of software architectures would greatly reduce the number of variant systems currently being utilized or developed and would allow greater utilization of reuse through more commonality of software systems. Architectures developed for specific domains could be developed across the various military Services and used to expand potentially reusable software resources by enlarging the domain envelop -- such as anti-aircraft missile technology which has application across all Services with such things as targeting and tracking software. Expanded domain architectures could easily provide exponential growth in lines of reusable code, and open up opportunities to utilize technology advances once realized only by individual Services.

c. Update the standardized SEI Software Development Process models to include software reuse. Adding software reuse to the model templates would provide a convenient guide to the time and method of inclusion of software reuse into the software development process. Such standardized templates would also serve to heighten awareness of software reuse.

Coordinated with the introduction of a reuse template should be the inclusion of software reuse into the software engineering life cycle. As described in chapter III, the software life cycle should provide ample opportunity to implement reuse at nearly every phase of the software life cycle. In order to obtain maximum effectiveness, it is critical that software reuse be integrated during the entire life cycle and not just during the development phase. The greater the focus on reuse, the wider the application and greater the potential for overall benefit.

d. Finally, once reuse has been addressed within the DoD, performance criteria for suppliers, distributors, maintainers based on use and customer satisfaction must be established. Without the support and full cooperation of the contractor side of the software development business, any DoD effort to implement reuse will be an exercise in futility.

Not only must DoD standards for reuse implementation be exported to the civilian software development community (primarily the contractors/ developers who are normally employed to develop DoD software) but acceptance of and conformity to these standards must also be utilized as criteria for contract award and measuring program progress.

2. Training and Education

a. The DoD must implement an extensive and intense training effort for DoD personnel in reuse techniques and utilization of available resources. Reuse methodologies and

techniques must be conveyed to those responsible for software development through the various curricula available through many of the DoD schools, organizations, and correspondence courses. Reuse training material should receive wide-spread dissemination while add-on correspondence courses in reuse technology and management must be made available and mandatory for Program Managers.

b. A reuse infrastructure must be established within the DoD. Essentially, a mechanism for sharing information and disseminating the latest technologies and methodologies must be developed to reach all aspects of software development and management. This infrastructure should include an annual software reuse symposium or conference and some form of newsletter to transmit information such as lessons learned, cost savings for programs, success stories, and legal issues concerning such things as licensing fees and restrictions. Such measures would serve to link the entire DoD software development effort, providing all developers and managers with a standardized and acceptable method for passing the most current and critical information in a timely and efficient manner.

c. The general awareness of the development community must be heightened with regard to reuse libraries and depositories. the user/development community must be provided with listings of available libraries, contents, cataloging systems, methods of utilization, and knowledgeable points of contact.

All the plans, libraries, depositories and regulations of the DoD cannot foster the integration of software reuse unless the developers are aware of the resources at hand. Only through a concerted effort to educate the PMs and their staffs on multiple levels and a host of reuse subjects can the DoD hope to implement widespread software reuse within the acquisition community.

3. Management

a. An effort should be made at the highest levels of DoD and industry to define and develop appropriate financial incentives and other promotionals for the production and use of quality reuse assets. It is clear that an undertaking of this magnitude will necessitate the best effort from both sides of the acquisition process. Because of the vast potential area of impact, high level representatives from all of the military Services, as-well-as NASA and other concerned agencies within the Government need to meet with industry leaders in software development and production as-well-as hardware producers to decide on the appropriate and most influential incentives for implementing reuse on a broad and standardized front. Current level of difficulty in instituting software reuse, combined with dubious returns, makes the use of incentives to inspire reuse all the more important. These incentives must appeal to the Program Manager and his staff as-well-as the commercial contractor and his developers. Participation by all of the interested or

concerned parties will be required in order to develop equitable yet inspirational incentives for utilizing software reuse. These incentives must apply not only to those who utilize reusable code, but those responsible for developing code intended to be reused, and hardware developers who must readily utilize open architectures which readily accept reusable software. Consequently, these incentives must be linked and dependent upon one another to be effective in encouraging reuse on a scale broad enough to impact all DoD/Government development efforts.

b. The standard development templates utilized in the acquisition process for software development must be revised to include reuse plans and strategies within the acquisition process. The establishment of a specific review process to determine the validity of inclusion of reuse in a program's development must be included in the acquisition approval process. This process should consist of incorporating specific approval points within the acquisition cycle which can only be passed after considering the pros and cons of including reuse into the program. It is imperative that such a program establish specific criteria which will force the PM to investigate and consider the potential for including software reuse into the program development.

c. A concerted effort must be made at the Service Acquisition Executive level or above to provide financial and technical support to PEOs and PMs for reuse initiatives.

[Ref. 38:p. 29] Any effort to push software reuse must be endorsed by and pushed from the top down.

This is probably best accomplished through the preparation and presentation of a business case to the Service Acquisition Executives to highlight the Service-wide advantages of implementing a reuse strategy for system acquisition. The case should identify potential domain areas for application of the proposed strategy, long-term (5 to 10 years) benefits to the various Services, and initiatives that are currently proceeding within each individual Service. Critical to the success of this effort should be the demonstration of the need for additional support (technical, financial, and moral) to the PEOs and PMs during the start-up phase of any reuse approach. It should be shown that relatively small amounts of money can make the difference between success and failure for the initiation of a reuse program, and that this money should be used to support the initial set-up and maintenance costs for library capability, the definition of common requirements and domain-specific architectures, and the additional development costs of reusable assets to support implementation. [Ref. 38:p. 20]

d. Finally, a DoD evaluation of current programs should be performed to collect data on all major programs incorporating reuse. These programs should be examined to determine "what works and what doesn't" with respect to implementing reuse into development efforts. This examination

would provide valuable information to other efforts such as the area of requirements and domain standardization. This could also open the development process to incorporation of an evaluation system of program performance that would include software reuse. Such an evaluation should examine the PM's conformance with the reuse plan, accomplishment of reuse goals, effective use of tools, and identification and resolution of problems in reuse.

4. Libraries

a. The neglect that current software libraries have suffered must be corrected by developing library standards, interface specifications, certification and acceptance criteria, library network interconnections, library interoperability, and configuration management. This will require close coordination with program/project offices to determine actual requirements, test ease of utilization, availability and access, and to develop quality assets. Again, this needs to be both a multi-Service and multi-domain project to prevent repetition of the poorly regulated, disjointed and generally neglected system currently in place.

b. Utilizing the information gathered in the process described above, domain specific libraries should be developed and populated. These libraries must be cross-Service, open architecture facilities, with logically conceived cataloging systems to provide easy access to domain specific and relatively risk free software to development projects.

Drawing on the PM experience and expertise in the development process, libraries should become considerably more valuable as a development resource. [Ref. 38:p. 24]

5. Legal and Contractual Issues

a. Of critical importance to any reuse effort will be the establishment of a mechanism for removing barriers for software development companies to legally utilize software produced from other software development companies. Current restrictions on utilization of proprietary software and architectures must be breached and equitable guidelines established to facilitate fair compensation for use of reusable code by other developers. Areas which must be addressed range from liability assessment for DoD programs damaged or delayed by reusing poor quality software, to responsibility for maintenance and upgrade of software cacheted in libraries, to royalties for the initial reuse of software and subsequent utilization of programs containing reused software developed by a second party.

b. Changes in the legal parameters of the acquisition process, allowing financial incentives based on performance criteria for implementation of reuse, should be developed and instituted for suppliers, developers, and maintainers. This approach should include contractor clauses to support reuse and evaluation criteria for RFPs. With reductions in DoD acquisition expenditures driving reductions in the number of firms willing to compete for defense contracts, financial

incentives may prove the necessary enticement to keep current contractors interested and bring new competitors into business with the DoD while simultaneously integrating software reuse into the development process. This would provide not only incentive to reuse software, but allow contractors to collect repeatedly for development efforts utilizing previously developed software.

C. LEVELS OF EXPERTISE AND KNOWLEDGE

Finally, that prevalent yet intangible human element which enters into and impacts the daily business of weapon system development must be addressed. Each level of effort within the development process (division, product, program) must be addressed. The level of knowledge required by the hardware or software division chief and his personnel is much greater and more detailed than the technical expertise required at the PM and DPM level. Because of these differing requirements for education and information at the various levels, efforts to address the "human factor" must be tailored to the specific area of effort. Any effort to dispel the disbelief or skepticism held by those involved in software development must be dealt with on a level which will specifically address those particular problems and ideas.

The DoD must apply a broad-based education program to all levels of effort involved in program development. The lowest level needing attention is at the division level within each PM office. This is the most technical level and accordingly

will require the most in-depth and detailed effort. The PM and DPM levels, being more management oriented, will not require such formal or detailed education in the actual mechanics of reuse. However, at the PEO level, the educational process must encompass everything from the broad-based overview to the detailed functional mechanics of reuse.

This educational effort must be both formal and informal. Formal classes must be given at all levels to educate development and management personnel on software development models and templates. These classes must be focused on specific levels of program development operations, depending on the particular audience. This instruction must include identification of the opportunity points in the development process to initiate or institute software reuse and the methods required to import reusable software into the target program. Course material must also cover procedures to implement thorough documentation techniques to facilitate integrating new code into software libraries. The education process must also include overviews of programs sharing particular weapon system application domains. These domains must include both inter- and intra- Service applications in order to maximize the benefit of reuse. Again, the level of detail and complexity must be tailored to the audience and its part in the development process.

Informal working groups, update and refresher sessions, and in-house seminars must be conducted on a regular basis.

These informal tools should cover the development and release of new reuse tools and templates within the DoD. Such a forum would also serve as a vehicle to pass information about concurrent development efforts which may have reuse application or affect software development in other programs. This is also the ideal mechanism to promote the reuse library system by regularly disseminating information on newly annexed code modules and documentation.

As mentioned above, the focus at the PEO level must cover the gamut of instruction, from general overview to detailed software reuse implementation procedures. The Program Executive Office level of operation is critical to the successful execution of any reuse program, regardless of the level of application. The PEO must serve as the integrator of reusable software into subordinate program domains and as the primary point of contact for software development with other PEOs. The PEO should also exercise executive control over reuse efforts within the specific program domains in order to pass critical development information in a timely manner and provide additional expertise to the PMs when necessary.

D. SUMMARY

Overall, the effort to implement software reuse on a scale that will produce acceptable returns on investment and productivity must be pushed on many fronts and at many levels. It should be apparent that most of the inhibitors and barriers can and must be assaulted simultaneously. Because so many of

them are linked together, often crossing numerous category boundaries in scope and impact, any effort to address one barrier or inhibitor may simultaneously affect another. The obvious danger is advertently creating more problems in associated or related areas through the application of faulty solutions or poor execution of good solutions. Consequently, all efforts to overcome these problems must be closely coordinated and controlled.

The solutions offered above are only some of the actions necessary to initiate reuse at the DoD level. These solutions range from the esoteric and technical, to the simplistic and comm sensical. Obviously, there are far more approaches to the problem than can be described here. However, all attempts at overcoming these barriers will have an impact, either positive or negative, on the Army wide implementation of software reuse. As these plans are implemented, other courses of action will become evident and further solutions will present themselves. Only time will tell which of these efforts will be successful and which will be shunted into obscurity. But one thing is clear, the first step must be to convince those responsible for software development that reuse is the key to future program success. The more for less military is now a reality. Excuses for inefficiency can no longer be tolerated, the cost in terms of time, money, and successfully fielded systems is just too high.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. INTRODUCTION

It should be obvious by now that the U.S. Army has begun to address the multitude of problems associated with the development of software for automated weapon systems. Facing a staggering demand for sophisticated, highly automated weapon systems requiring greater productivity from the limited and available software-producing resources, and facing severely shrinking defense expenditures, the military has been forced to relook its efforts to meet these needs. The different Services have pursued or are pursuing various individual and joint solutions to the imminent software shortage. However, of the currently available technologies and methodologies available to attack this problem, the method with the greatest potential return of investment is software reuse.

B. THE WORK AT HAND

The Department of Defense has embarked on a host of different programs to explore this option, with the Army taking the lead on several. In response to pressure from Congress, the overall lead for the Army effort has been tasked to the Army Software Reuse Council and Working Group. Leading the Army's effort in applying software reuse are the RAPID software library program and the AFATDS and ATCCS software/hardware development programs.

This Reuse Working Group is attempting to consolidate the entire Army software reuse effort in order to capitalize on economies of scale offered by a widespread program and the obvious advantages of utilizing multiple domains and their potential contributions and applications. Current Army reuse efforts are focused on laying the foundation or ground-work infrastructure to employ reuse on a Service-wide basis once all the component management and resourcing assets are in place.

The current plan requires that the Army implement the following functions:

1. Specify currently existing domains within the Army and identify criteria necessary to prioritize, select and qualify these domains for reuse application [Ref. 39:p. 3].

2. Define potential reuse products within these selected domains, to include domain analysis output and software components generated by the development life cycle, as-well-as component validation criteria for new applications [Ref. 39:p. 3].

3. Determine ownership criteria for new and reused components [Ref. 39:p. 3].

4. Implement changes in the current acquisition process to provide for the consideration and evaluation of reuse during the entire acquisition life cycle [Ref. 39:p. 3].

5. Consider and evaluate proposed deviations from the development process to integrate and utilize reusable components during each phase of the development life cycle [Ref. 39:p. 3].

6. Define and develop models as guides to business decisions related to reuse implementation [Ref. 39:p. 3].

7. Define reuse metrics and establish guidelines and procedures to govern collection and analysis of pertinent data [Ref. 39:pp. 3-4].

8. Define component standards and guidelines to describe characteristics and evaluation criteria necessary for reuse component certification and inclusion in reuse libraries [Ref. 39:p. 4].

9. Specify a technology base investment strategy to recognize likely technologies, designate and track reuse-related research and development efforts, and identify appropriate means of transitioning these efforts into actual practice [Ref. 39:p. 4].

10. Describe necessary training and education to impact all levels of development decision making and influence both the internal and external environment to facilitate reuse [Ref. 39:p. 4].

11. Identify existing products and services for potential reuse exploitation [Ref. 39:p. 4].

12. Finally, monitor, assist, contribute to , and extract lessons learned from concurrent software reuse efforts by other organizations both inside and outside the Government [Ref. 39:p. 4].

C. OVERCOMING THE BARRIERS

These steps are necessary to overcome the myriad barriers and inhibitors which currently frustrate wide-spread Army implementation of software reuse. These barriers are generally catalogued into five broad categories. First, and maybe most far reaching, is the lack of standards in both the hardware and software development processes, associated development tools (metrics), and architectures. The second category is poor or inadequate training and education for those most able to influence software development decision making. No less important, is the third category, management. The lack of adequate reuse incentives for both the decision-

maker as well as the contractor, a situation compounded by lack of trade-off mechanisms, cost models, experience, and vision provide new frontiers and challenges in software development management. The fourth category, and possibly the one of greatest immediate impact to the implementation (or lack thereof) of reuse, is the lack of any centralized catalog of assets. The lack of adequate and logically cataloged resources, plus inefficient means of surveying and retrieving the currently available resources, has prevented any substantial attempt at reuse. The last category is probably the most restrictive of the five. The fifth category is that engaging the morass of legal and contractual issues associated with patents, copyrights, and proprietary data rights. Because of the complex issues and economic implications involved in this category, it is ripe for long term dispute, arbitration, and litigation, and may prove to be the most difficult to competently and conclusively overcome.

Although these reuse barriers and inhibitors are classified into five separate categories, all are related and generally bear influence on the others. Any attempt to overcome or solve any particular problem in one category will necessarily cross into another category. Obviously, simple answers or solutions will not be adequate to thwart these barriers. Only bold, forceful, deliberate action to implement reuse will prove successful against these overwhelming impediments.

Faced with these seemingly insurmountable obstacles, the Program Managers, as a group, have rejected the prospect of voluntarily integrating software reuse into the software/hardware development process. But, most PMs, being military officers, admitted they would "soldier on" with reuse if necessary. The consensus of those interviewed was that reuse should not be applied to current programs, but should be a "ground-up" proposition for new programs. All personnel interviewed believed that software reuse will become a reality within the Army in the near term. Those within the program office, from PMs to DCs, believed also, that PMs should have sufficient technical background in software and/or hardware to be able to work competently with incoming reuse technologies. Many of those interviewed were skeptical of the technical skills displayed by more than a few of the currently serving PMs.

Finally, all those interviewed believed that implementation of software reuse could only become a reality if the program were top driven. Everyone suggested that essential aspects of reuse implementation, such as reuse library service, software porting studies, and reuse administration and documentation, be at a sufficiently high level to span all potential contributing sources within the Army (and/or the DoD). Those interviewed recommended that whatever organization the Army develops to provide overall operational control of software reuse, it must be able to tie related PMs

together, or at least be able to interface with the PEOs and integrate outside resources and information into subordinate programs.

As stated before, the human element is the most deep-seated barrier facing software reuse implementation. The resistance to change is a powerful, institutionalized force and will be difficult to overcome. Only through the liberal application of time and education can this inhibitor be eliminated.

Both technical and human barriers must be overcome to successfully implement this program at either the Army or DoD level. Human resistance will be conquered through education. The technical barriers, however, will be overcome only through a concerted, systemic effort on a broad front. The most promising approach to implementing reuse is an Army/DoD multilevel program currently being launched. The program consists of implementing several initiatives spanning standardization, education and training, management, libraries, and legal and contractual issues. These initiatives seek to integrate new and emerging technologies with tools, methodologies, and templates currently being utilized in software development. The Army and DoD are working jointly to take advantage of lessons learned through multiple software reuse efforts and experiments. Networks are being established to disseminate reuse information among interested organizations in all military Services and selected

Government organizations. And, possibly most important, serious reform of the acquisition process with respect to software and automated hardware development, is currently being reviewed and revised. This acquisition review includes mandatory consideration and, in some cases, inclusion in all software acquisition and development programs and some hardware development programs.

D. RECOMMENDATIONS

A great deal of attention has been placed on the organization necessary to facilitate software reuse. A program without a strong and effective organizational structure has no hope of success. All parties involved in program management, software development, and software reuse agree that this organization must have the following characteristics:

1. It must be either a DoD level organization to ensure it is powerful enough to impose directives and controls on subordinate developmental organizations and programs, or it must be such that it is a separate organization mirrored within all military services, tied through administrative channels to pass information, development techniques, and program progress.

2. The organizations must oversee all aspects of software development, from software repositories and libraries, to template and model development and dissemination, to administrative and regulatory management of software development and acquisition.

3. It must serve to link each particular program with each of the other programs of similar domain or software requirements, or at least link the governing PEOs of the affected programs.

4. It must bear some of the decision making responsibility for source selection, development strategy, and program technical reviews.

5. And finally, it must be adequately funded to ensure sufficient resources and organizational clout to affect the direction of development programs in all effected areas.

The wheels of change are currently turning with respect to the creation of such an organization. It can only be hoped that this organization can benefit from the synergy resulting from the interaction of the strategic issues mentioned herein. The synergistic effect should ultimately reduce the time required to field critical systems, improve quality, and reduce costs and risk associated with software-intensive systems [Ref. 39:p. 4].

E. AREAS FOR FURTHER RESEARCH

The issues, organizations, solutions and models mentioned here are not in themselves sufficient to overcome the time and expense necessary to bring a sophisticated, automated weapon system to successful production and field implementation. A great deal of further study is needed. Even as the aforementioned solutions are being implemented and controlling organizations formed, more questions are being raised and new reuse inhibitors are being discovered. During the course of this research, several related problems were uncovered. The following areas offer opportunities for further study and will necessarily need to be addressed in the course of Army software reuse implementation.

1. Examine potential software repository standardization and structure, to include domain classification, retrieval and donation systems, with the object of arriving at an optimal organizational and administrative structure.

2. Examine potential organizational structures and program interface mechanisms for a DoD level or Service level software reuse advocate/control organization to oversee implementation of software development and reuse across a broad spectrum of applications and domains.

3. Explore recent and potential improvements in software development templates and models utilizing the latest software reuse technologies.

4. Examine the contractor's managerial problems, inhibitors, and barriers in developing and implementing software reuse, and potential Government incentives which could be offered to contractors to overcome these barriers.

5. Examine current DoD and Army software development and acquisition policy for potentially advantageous changes which might incorporate or implement software reuse as part of the program development and review process.

6. A thorough examination should be conducted to study the legal inhibitors and barriers from both the Government and contractor perspectives, with an objective to find legal work-arounds which will foster greater contractor participation in developing and utilizing reusable software.

While these offered areas of further study a certainly not a complete list of potential subjects, it does cover some of the more difficult, non-technical issues which must be overcome before the DoD is able to successfully implement software reuse on a scale that will provide economic dividends for the developing agency. Much work remains to be done, especially in the area of convincing those most responsible for developing and utilizing reusable software that it is a worthwhile idea. However, even as the program is being

inaugurated at the DA level, work continues at all levels to win the software reuse battle.

APPENDIX A

PROGRAM OFFICE QUESTIONNAIRE

The following questions were asked of selected Army Program Managers as a means of acquiring information for this thesis. Cooperative responses to initial questionnaires were given a second, follow-up questionnaire, or in some instances, visited by the author to interview the PM and his staff. The second questionnaire was targeted at not only the Program Manager, but the DPM and the subordinate program Division Chiefs in an attempt to get a more well-rounded picture of PM operations.

A. Initial Questionnaire

1. Does your project require software development or acquisition as part of the program?

2. Is a significant amount of your program budget and time devoted to software development issues? If so, ho much?

3. Is your project software developmental or is it an upgrade of currently utilized or existing software?

4. Does your system software share utilization with any other system under development or currently in the field?

5. Does your system software have potential to be utilized by other systems?

6. Do you know of any other system currently being developed or already fielded which might have software that could be utilized by your system?

7. Does your project office engage in any form of joint software development effort with any other project?

8. Is there any exchange of information concerning software problems and development with projects and programs either inside or outside your own PEO structure?

9. Has your project software development affected the cost or schedule of your program?

10. Are you familiar with the current effort by DoD to initiate a Software Reuse Program?

11. Would your project software development benefit from a centralized library of existing programming and cataloging of developmental efforts?

12. Do you think future programs would benefit from such a centralized library?

B. Follow-up Questionnaire

The intended target of any particular question is given in parenthesis before each question. In some instances, the questions are the same as those found in Questionnaire One, but are intended to illicit information from different individuals within the PM office. Those interviewed or surveyed included Program Managers, Deputy Program Managers, Hardware and Software Development and Management Division Chiefs, and in some cases, technical specialists.

Although the initial questionnaire was intended to be answered by the PM, in some instances, the DPM responded. Consequently, some of the follow-up questions are intended to be answered by either the PM or DPM, depending on who responded to the initial questionnaire.

1. (PM/DPM) How much control do you exercise on a daily basis within your organization concerning software development?

2. (All) What is your background with respect to hardware and software technology, engineering, and development methodologies?

3. (PM/DPM) How competent are your subordinates (Deputy PM, Division Chiefs, etc.) with respect to software and hardware development and the latest emerging technologies, including software reuse?

4. (All) How open are you lines of communication inside your organization?

5. (All) How open are your lines of communication to organizations outside your program, such as the PEO, other programs within the PEO, and programs outside the PEO?

6. (DC) How knowledgeable are the PM and DPM in matters of software/hardware development and emerging technologies such as reuse?

7. (DC) How much direct control or influence does the PM/DPM exercise over your division and its efforts on a daily basis?

8. (All) Does your system software share utilization with any other system under development or currently in the field?

9. (All) Does your system software have potential to be utilized by other systems?

10. (All) Do you know of any other system currently being developed or already fielded which might have software that could be utilized by your system?

11. (All) Are you familiar with the current effort by DoD to initiate a Software Reuse Program?

12. (All) Would your project software development benefit from a centralized library of existing programming and cataloging of developmental efforts?

13. (All) Do you think future programs would benefit from such a centralized library?

APPENDIX B

DEFINITIONS

Ada
A comprehensive, Pascal-based programming language developed for the DoD to implement both business applications and embedded applications (such as rocket guidance systems).

Application Software
Higher order language programs that can perform specific, user-oriented tasks.

Architecture
A structural concept for a complex software application and a definition of the software components that populate the structure.

Assembly Language
Computer languages that allow the use of mnemonic names for machine language instructions and operands in the place of the binary machine codes.

Automated Weapon System
Weapon system utilizing and/or incorporating digital information system(s) as an integral part of either the actual weapon or its command and control element or both. The computer hardware and software in an automated weapon system perform such tasks as input/output control, system diagnostic functions, and program function execution such as acquiring, tracking, and closing on designated targets.

Bit
A unit of information storage capacity, as either of the binary digits 0 or 1 in a computer memory. [B(inary)(dig)IT]

Critical Path
The longest path through a project from beginning to end. It includes all the activities that, if delayed, would stall the project's completion.

Do Loop
A program execution command used o implement repetitive operations within an executable function.

Domain
A specific phase or area of the software life cycle in which a developer works. Domains define developers and users areas of responsibility and the scope of possible relationships between products.

Domain Analysis
The process of identifying the preliminary requirements for software parts to fill common needs of a selected application domain through an intensive examination of applicable architectures. The process consists of developing a preliminary model and classification scheme for the domain and refining the model and identifying common objects, structures and function which are candidates for reusable software parts.

Embedded Computer System
A computer system as an integral part of a larger system such as a weapon system or communication system which cannot be separated or deconstructed into functional component parts without degrading or incapacitating the system capabilities.

Granularity (of software components)
Granularity refers to the number and size of software modules or sub-components which compose the varying parts of a software program. Essentially, the smaller the modules and their component parts, the greater the software granularity.

Hardwired
An industry term referring to hardware solutions to software application problems. Essentially, this term refers to physical work-arounds through electronic engineering applications utilized where software solutions are not practical or possible.

Higher Order Languages
Computer programming languages utilizing statements which typically resemble mathematical formulas or English expressions much more than assembly language commands. Examples of HOLs include FORTRAN, COBAL, CMS-2, and Ada.

Instruction Set Architecture
The internal and fixed repertoire of instructions that compose the required integration pathways describing the hardware/software interface.

Life Cycle
The stages and processes through which software passes during its development. This includes requirements definition, analysis, design, coding, testing, and maintenance.

Machine Language
The binary codes which are understood directly by the computer hardware.

Metrics
Quantitative analysis values calculated according to a precise definition and used to establish comparative aspects of development progress, quality assessment or choice of options.

Mnemonic
A symbolic name used in a computer program rather than an actual numeric address.

Operand
A field specifying where in the computer the data for a particular operation or function is located.

Software
The combination of computer programs and documentation needed to cause computer hardware to perform a certain task or tasks.

Software Bindings
A software component that provides an application with a means to access other software written in a language different from that of the application.

Software Component
This refers to named modules of reusable information that can be manipulated by a target reuser. Software components may be further decomposed into other components and software units.

Software Module
The discrete components of a software program. Each module is separate and distinct from other modules such that a change in one component has minimal impact on other components.

Software Object
A software system element with State, Behavior, and Identity. Similar objects are grouped in a single Class or Subclass.

Software Part
A piece of data resulting from some phase of the software life cycle. Examples include: executable code, a requirement specification, an interface specification, and a data flow graph design. A software part can be catalogued as a collection or hierarchy of smaller parts.

Software Portability
The ease with which software can be transferred from one computer system or environment to another.

Software Reuse Technologies
Technologies, tools, and programs developed to foster software reuse. These are generally innovative, leading edge approaches that can include both software and hardware systems.

Software Spoilage
The portion of a software program that is lost or damaged or does not readily present itself to software porting.

Software Unit
Any logical set or groupings of instructions to a computer, such as a module or package.

Taxonomy
The science, laws, or principles of classification.

APPENDIX C

ACRONYMS

ADAS	Architecture Design and Assessment System
AFATDS	Advanced Field Artillery Tactical Data System
AR	Army Regulation
ASSET	Asset Source for Software Engineering Techniques
ATCCS	Army Tactical Command and Control System
BFA	Battlefield Functional Area
CAMP	Common Ada Missile Packages
CARDS	Central Archive for Reusable Defense Software
CASE	Computer-Aided Software Engineering
CECOM	Communications-Electronics Command
COTS	Commercial Off-The-Shelf
DA	Department of the Army
DARPA	Defense Advanced Research Projects Agency
DC	Division Chief
DISA	Defense Information Systems Agency
DoD	Department of Defense
DOS	Disk Operating System
DPM	Deputy Program/Project/Product Manager
DSMC	Defense Systems Management College
FAR	Federal Acquisition Regulation
FY	Fiscal Year
GOTS	Government Off-The-Shelf
HOL	Higher Order Language

IDA	Institute for Defense Analyses
JIAWG	Joint Integrated Avionics Working Group
JLC	Joint Logistics Commanders
NASA	National Aeronautics and Space Administration
NDI	Non-Developmental Item
NSIA	National Security Industrial Association
PDSS	Post Development Software Support
PEO	Program Executive Officer
PM	Program/Project/Product Manager
RAPID	Reusable Ada Products for Information System Development
SCSI	Small Computer Systems Interface
SEI	Software Engineering Institute
SLOC	Software Lines of Code / Source Lines of Code
STARS	Software Technology for Adaptable, Reliable Systems
VHSIC	Very High Speed Integrated Circuit

LIST OF REFERENCES

1. U.S. Department of the Army. Army Acquisition Executive Support Agency. "Life Cycle Software Engineering Centers." Army Research, Development, and Acquisition Bulletin, (Nov-Dec 1990), pp. 26-29.

2. Biggerstaff, Ted J., and Alan J. Perlis, editors. 1989. Software Reusability. Vol. 1, "An Expansive View of Reusable Software," by Ellis Horowitz and John B. Munson. New York: ACM Press (Addison-Wesley Publishing Company).

3. Distaso, Jack R. 1989. "Ada: Experience and Trends." In Managing Software Into the '90s... Acquiring the Competitive Edge: Proceedings of the Army Software Conference in Eatontown, New Jersey, by the U.S. Army Communications-Electronics Command, p. 124.

4. U.S. Department of the Army. Office of the Secretary of the Army. Strategic Force, Strategic Vision for the 1990s and Beyond, by the Honorable Michael P. W. Stone and General Gordon R. Sullivan. A Statement on the Posture of the United States Army, Fiscal Year 1993. Washington, DC, U.S. Government Printing Office.

5. Konda, Suresh, Patrick D. Larkey, and W. Gary Wagner. National Software Capacity: Near-Term Study. Technical Report produced by the Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, May 1990. DTIC, CMU/SEI-90-TR-12.

6. Department of Defense. Defense Systems Management College. Mission Critical Computer Resources Management Guide (1990). Washington, DC, U.S. Government Printing Office.

7. Pasztor, Andy. "U.S. to Upgrade, Not Replace, Its F-16 Fleet." Wall Street Journal. 9 April 1992. sec. A, p. 5.

8. Kitfield, James. "Is Software DoD's Achilles' Heel?" Military Forum. Washington, DC, U.S. Government Printing Office. (July 1989), pp. 27-35.

9. "Suggestions for DoD Management of Computer Software." Managing Software: The Journal of Defense Systems Acquisition Management 5 (Autumn 1982).

10. Boehm, Barry W. and Thomas A. Standish. "Software Technology in the 90's: Using an Evolutionary Paradigm." Computer 16 (November 1983), pp. 29-32.

11. Coles, R. J., et. al. "Software Reporting Metrics." A Technical Report by Mitre. Washington, DC (November 1985).

12. Blanchard, Benjamin S. and Wolter J. Fabrycky. Systems Engineering and Analysis. 2d ed. Englewood Cliffs, NJ: Prentice-Hall, 1990.

13. Arthur, Lowell J. Measuring Programmer Productivity and Software Quality. New York: John Wiley & Sons, 1985.

14. DeMarco, Tom. Controlling Software Projects: Management, Measurement, and Estimation. New York: Yourdon Press, 1982.

15. U.S. Department of Defense. "Software Documentation." Proceedings of the Joint Logistics Commanders' Joint Policy Coordinating Group on Computer Resource Management in Monterey, CA, November 1981.

16. U.S. Department of the Army. Program Executive Office: Strategic Defense. "Reusable Software Acquisition Environment Guidebook for the STARS Program." A Technical Report prepared for UNISYS Defense Systems by DSD Laboratories, Inc., Sudbury, MA, (22 Nov. 1991).

17. Baker, Brian, and Anna Deeds. "Industrial Policy and Software Reuse: A System Approach." In Reuse in Practice Workshop at the Software Engineering Institute, Pittsburgh, PA, July 11-13, 1989.

18. U.S. Department of the Army. U.S. Army Communications-Electronics Command. "The Business Issues Associated with Software Reuse." A report of a study conducted by the National Security Industrial Association's Software and C3IC Committee's Joint Sub-Committee on Software Reuse. 15 December 1990.

19. U.S. Department of the Army. Program Executive Office for Command and Control Systems. "Mission Statement for PEO Command and Control Systems." Prepared by the Army Acquisition Executive Support Agency, Alexandria, VA, June 1991.

20. DoD Instruction 5000.2, "Defense Acquisition Program Procedures," February 23, 1991. Washington, DC, U.S. Government Printing Office.

21. DoD Directive 7920.1, "Life Cycle Management of Automated Information Systems (AIS)," October 17, 1978. Washington, DC, U.S. Government Printing Office.

22. DoD Directive 3405.1, "Computer Programming Language Policy," April 2, 1987. Washington, DC, U.S. Government Printing Office.

23. DoD-STD-2167A, "Defense System Software Development," February 29, 1988. Washington, DC, U.S. Government Printing Office.

24. U.S. Department of the Army. U.S. Army Signal Center. "Information Sheet: Tactical Automatic Switch (TAS) AN/TTC-38," May 20, 1987. Fort Gordon, GA.

25. U.S. Department of the Army. U.S. Army Signal Center. "Applications Guide: The AN/TTC-39 Circuit Switch," August 18, 1986. Fort Gordon, GA.

26. Grier, Peter. "DoD's Computer Hardware Paradox." Military Forum. Washington, DC, U.S. Government Printing Office. (July 1989), pp. 36-40.

27. Arango, Guillermo Francisco. "Domain Engineering for Software Reuse." PH.D. dissertation, University of California (Irvine), 1988.

28. Jones, T. Capers. "Reusability in Programming: A Survey of the State of the Art." In Programming Productivity: Issues for the Eighties. Washington, DC: IEEE Computer Society Press, 1986, pp. 372-377.

29. U.S. Department of the Army. Program Executive Office for Aviation. Software Reuse Rationale. A technical report generated by the Joint Integrated Avionics Working Group, St. Louis, MO. May 26, 1989.

30. Druffel, Larry E., Samuel T. Redwine, Jr., and William E. Riddle. "The DoD STARS Program." Computer 16 (November 1983), pp. 9-11.

31. Roberts, Russel. "The "RAPID" Way to Software Development." Army Research, Development, and Acquisition Bulletin. (July-Aug 1991) pp. 31-32.

32. Palmer, Constance. "Reuse in Practice." Position paper presented during the Reuse in Practice Workshop, in Pittsburgh, PA, July 11-13, 1989, by McDonnel Douglas Missile Systems Company, 1989.

33. Donaldson, Cammie. "Knowledge-Based Reusable Software Synthesis System in NASA." In Software Reuse Issues -- Workshop Proceedings (Conference Publication 3057) 17-18 Nov 1988, by NASA Langley Research Center.

34. Voigt, Susan, and Carrie Walker. "Reuse Research Plans at Langley Research Center." In Software Reuse Issues -- Workshop Proceedings (Conference Publication 3057) 17-18 Nov 1988, by NASA Langley Research Center.

35. Bishop, Dr. Peter C. "Technology Transfer in Software Engineering." In Software Reuse Issues -- Workshop Proceedings (Conference Publication 3057) 17-18 Nov 1988, by University of Houston at Clear Lake.

36. Department of the Army. Program Executive Office for Command and Control Systems. ATCCS Primer. An information pater published by PM CHS, Fort Monmouth, NJ, Summer, 1991.

37. Rubey, Raymond J. Software Management Guide. Hill Air Force Base, UT: Software Technology Support Center, April 1992.

38. Minutes of the DoD Reuse Impact Team. By Harry F. Joiner, Chairman. Fort Monmouth, NJ, March 3-4 1992.

39. Army Software Reuse Plan (Strawman). Plan produced by the Army software Reuse council and Working Group in response to a directive from the Senate Appropriations Committee. Provided by Stanley H. Levine, PM CHS, Fort Monmouth, NJ, Dec, 1992.

BIBLIOGRAPHY

Adams, Susan. Sun Catalyst Porting Reference Guide. Mountain View, California: Privately printed, 1991.

Berube, Margery S., Diane J. Neely, Pamela B. DeVinne, eds. The Armerican Heritage Dictionary, 2d ed. Boston: Houghton Mifflin Company, 1982.

Boillot, Michel. Understanding FORTRAN, 2d ed. New York: West Publishing Company, 1981.

Bowes, Robert J. "Technology Transition Reuse Acquisition Issues." Notes from a presentation to Software Reuse Working Group. Fort Monmouth, New Jersey. (3 Dec 1991).

Ehrlich, Eugene, Stuart Berg Glexner, Gorton Carruth, Joyce M. Hawkins, eds. Oxford American Dictionary. New York: Avon Books, 1982.

Horowitz, Barry M., Phd. The Importance of Architecture in DOD Software. Bedford, Massachusetts: Mitre Corp., 1991.

Hughlett, Eric C. "A Framework for Software Development." M.S. Thesis, Naval Postgraduate School, 1990.

Jones, Wilbur D. Jr., ed. Glossary: Defense Acquisition Acronyms and Terms, 4th ed. Fort Belvoir, Virginia: Department of Defense, Defense Systems Management College, 1989.

Laird, Charlton. Webster's New World Thesaurus. New York: Warner Books, Inc., 1982.

Schildt, Herbert. DOS Made Easy. Berkeley, California: Osborne McGraw-Hill, 1988.

TRW Systems Engineering and Development Division, Space & Defense Sector, Common ACCS Support Software (CASS) Development. Final V0.4 Software Evaluation Report: Workstation Management Block. Redondo Beach, California: TRW, Inc. 31 March 1992.

Turban, Efraim. Decision Support and Expert Systems. New York: Macmillan Publishing Company, 1990.

U.S. Army Communications-Electronics Command, Center for Software Engineering, Software Technology Division. Survey of Available Ada Bindings. Fort Monmouth, New Jersey: U.S. Army Printing Office, 1991.

U.S. Department of the Army. U.S. Army Signal Center. Applications Guide: The AN/TYC-39 Circuit Switch. Fort Gordon, Georgia: U.S. Army Printing Office, 1986.

U.S. Department of Defense. DOD Directive 5000.1: Major and Non-Major Defense Acquisition Programs. [Washington, D.C.]: U.S. Department of Defense, 1991.

U.S. Department of Defense. DOD Instruction 5000.31: Interim List of DOD Approved High Order Programming Languages (HOL's). [Washington, D.C.]: U.S. Department of Defense, 1976.

U.S. Department of Defense. DOD Standard 2168: Defense System Software Quality Program. [Washington, D.C.]: U.S. Department of Defense, 1988.

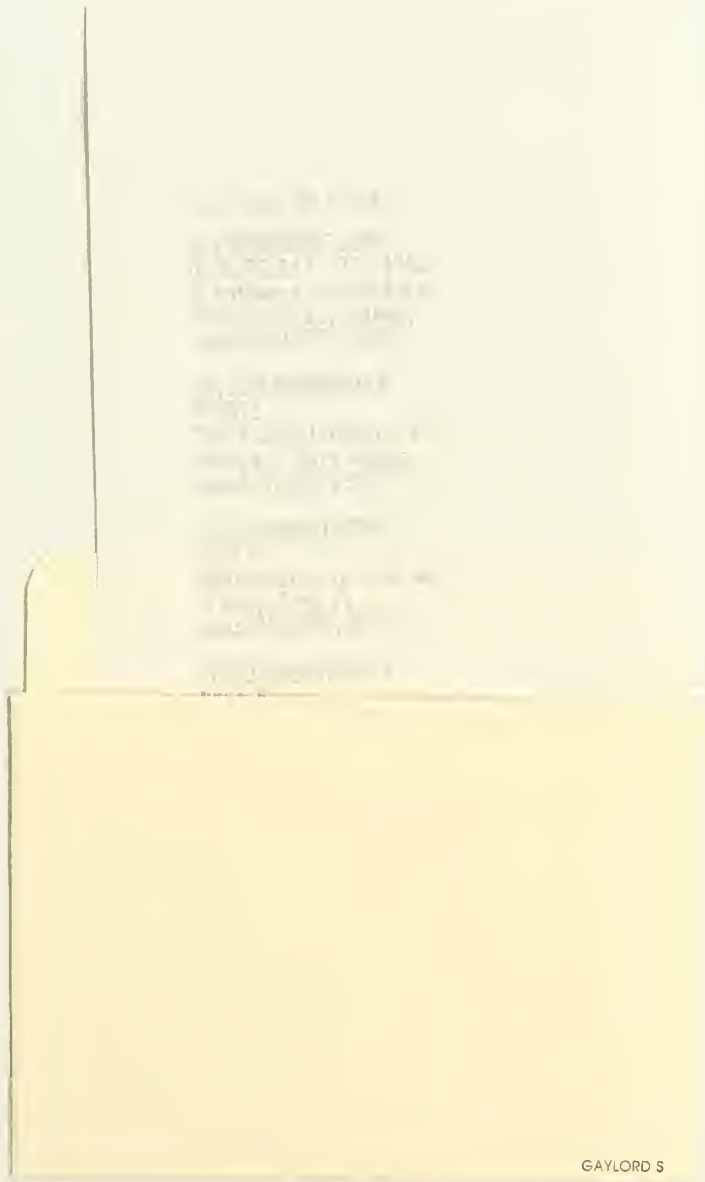
U.S. Department of Defense, Department of the Army. Army Regulation 73-1: Test and Evaluation Policy. [Washington, D.C.]: U.S. Department of Defense, Department of the Army, 1992.

U.S. Department of Defense, Department of the Army. Department of the Army Pamphlet 73-1: Test and Evaluation Procedures and Guidelines, Vol. 6, Software Test and Evaluation Guidelines (Draft). [Washington, D.C.]: U.S. Department of Defense, Department of the Army, 1992.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Prof. David V. Lamm (Code AS/Lt) Naval Postgraduate School Monterey, California 93943-5000	2
4.	Prof. Ronald A. Weitzman (Code AS/Wz) Naval Postgraduate School Monterey, California 93943-5000	1
5.	CPT Donald F. Burns III 135 Malibu Dr. Eatontown, NJ 07724	2

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101



GAYLORD S

3 2768 00312931 3